FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Analysis and Aggregation of Vulnerability Databases with Code-Level Data

Pedro Leite Galvão

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Carlos Viegas Martins Bispo

October 31, 2022

Analysis and Aggregation of Vulnerability Databases with Code-Level Data

Pedro Leite Galvão

Mestrado Integrado em Engenharia Informática e Computação

October 31, 2022

Abstract

The use of open source libraries and frameworks has become a widely adopted practice and many software projects nowadays, including commercial applications, have a large portion of its source code based on open source libraries. This practice is beneficial in terms of reducing the cost and increasing the speed of development, but has its costs in terms of security.

Many sources indicate that the number of disclosed vulnerabilities in open source packages is increasing every year, and attackers often target the software supply chain instead of the code of specific applications. The OWASP Top 10 lists Vulnerable and Outdated Components as the sixth major security risk for web applications. These problems make it necessary for software developers to manage and maintain the security of the dependencies in the projects in which they work.

To facilitate handling this problem, many solutions have been proposed both in academic literature and in the form of commercial products. There are several vulnerabilities scanners that rely on open source vulnerability databases to extract the information necessary to their operation.

Of particular interest is the case of vulnerability databases with code level information, because this kind of information can, among other applications, allow vulnerability scanners to identify the location of the vulnerability in the source code, or train machine learning models for recognition of patterns in source code that usually indicate the presence of vulnerabilities.

Project-KB is one of such databases in which code level information is available. Each entry contains a list of commits that fix the vulnerability. This enables the vulnerability scanner Eclipse Steady to identify vulnerable portions of code and analyse if they are reachable in a particular context. To guarantee the reliability of its data, entries in Project-KB are inserted only after manual analysis. The integration of different datasets would be beneficial to the operation of this project as well as for applications and studies that rely on it.

In the present study we provide an objective analysis of different vulnerability databases in terms of features, size, scope, as well as the reliability of the information they contain and determine the possibility of expanding Project-KB using some of these sources.

Keywords: Open source software, software vulnerabilities, code-centric vulnerability analysis

Resumo

O uso de bibliotecas e frameworks de código aberto tornou-se uma prática amplamente adotada e muitos projetos de software atualmente, incluindo aplicações comerciais, possuem grande parte de seu código-fonte baseado em bibliotecas de código aberto. Esta prática é benéfica em termos de redução de custos e aumento da velocidade de desenvolvimento, mas tem seus custos em termos de segurança.

Muitas fontes indicam que o número de vulnerabilidades divulgadas em pacotes de código aberto está aumentando a cada ano, e os invasores geralmente visam a cadeia de suprimentos de software em vez do código de aplicativos específicos. O OWASP Top 10 lista "Componentes Vulneráveis e Desatualizadas" como o sexto maior risco de segurança para aplicativos da web. Esses problemas tornam necessário que os desenvolvedores de software gerenciem e mantenham a segurança das dependências nos projetos em que trabalham.

Para facilitar o tratamento deste problema, muitas soluções têm sido propostas tanto na literatura académica quanto na forma de produtos comerciais. Existem vários scanners de vulnerabilidades que contam com banses de dados de vulnerabilidades de código aberto para extrair as informações necessárias à sua operação.

De particular interesse é o caso de bases de dados de vulnerabilidades com informações a nível de código, pois esse tipo de informação pode, entre outras aplicações, permitir que scanners de vulnerabilidade identifiquem a localização da vulnerabilidade no código-fonte, ou treinem modelos de aprendizado de máquina para reconhecimento de padrões em código-fonte que geralmente indicam a presença de vulnerabilidades.

Project-KB é uma dessas bases de dados que disponibiliza informações a nível de código. Cada entrada contém uma lista de commits que corrigem a vulnerabilidade. Isso permite que o scanner de vulnerabilidade Eclipse Steady identifique partes vulneráveis do código e analise se elas podem ser alcançadas em um contexto específico. Para garantir a confiabilidade de seus dados, as entradas no Project-KB são inseridas somente após análise manual. A integração de diferentes conjuntos de dados seria benéfica para o funcionamento deste projeto, bem como para aplicações e estudos que dependem dele.

No presente estudo, fornecemos uma análise objetiva de diferentes bases de dados de vulnerabilidades em termos de recursos, tamanho, escopo, bem como a confiabilidade das informações que elas contêm e determinamos a possibilidade de expandir o Project-KB usando algumas dessas fontes.

Keywords: Software open source, vulnerabilidades de software, análise de vulnerabilidades baseada em código

Acknowledgements

This work was developed as part of an internship in SAP Labs France, with the Open Source Security Analysis team.

Pedro Galvão

“Until I began to learn to draw,
I was never much interested in looking at art.”

Richard P. Feynman

Contents

List of Figures

List of Tables

Abbreviations

OSS	Open Source Software
API	Application Programming Interface
SCAP	Security Content Automation Protocol
CVE	Common Vulnerability and Exposures
CWE	Common Weakness Enumeration
CPE	Common Platform Enumeration
CVSS	Common Vulnerability Scoring System
CNA	CVE Numbering Authority
NVD	National Vulnerability Database
AST	Abstract Syntax Tree
CFG	Control Flow Graph
DFG	Data Flow Graph
SAST	Static Application Security Analysis
DAST	Dynamic Application Security Analysis
JVM	Java Virtual Machine
PyPA	Python Packaging Authority
GSD	Global Security Database

Chapter 1

Introduction

1.1 Context

Numerous software vulnerabilities are found and disclosed every year in open source projects. In this context, one of the main solutions for ensuring software security, is the use of automated vulnerability detection. Several tools were developed to respond to this necessity.

A particular area of interest is the detection of vulnerabilities in Open Source Software (OSS) libraries. A vulnerability in a software component can become exploitable in many of the projects that import this component. The fact that this information becomes visible to anyone makes it easy for attackers to exploit it. Very often, in spite of patches made to fix vulnerabilities, other projects continue to use outdated and vulnerable versions of the libraries containing them. This is what OWASP Top 10 indicated in 2021 as being the sixth major security risk for web applications [28].

Vulnerability Databases help with the task of identifying vulnerable dependencies and possible remediations to security issues. In this context, many different datasets have been formulated with different purposes, such as for feeding vulnerability scanners [30] and for training machine learning models for vulnerability detection.

The databases we will see in this study contain instances of vulnerabilities in OSS. Each entry of each database corresponds to a vulnerability found in a specific project and publicly disclosed. Each database provides different features associated to the vulnerabilities such as the project to which it belongs, the affected version, a classification, etc.

Of particular relevance for our study are the vulnerability databases that include code-level data about the vulnerability. This means data that enables the identification of which portion of code is vulnerable. As we will see, this kind of information can be used, among other applications, to increase accuracy of vulnerability scanners.

1.2 Motivation

This study was done in the context of an internship in Security Research in SAP Labs France. The company developed two products that are relevant for this study: Project-KB and Eclipse Steady.

Project-KB provides a set of vulnerabilities with code-level data whose reliability is assured by a process of manual verification. However it covers a relatively small amount of vulnerabilities when compared to some of the other databases, like NVD and OSV. The integration of other sources would allow tools and studies based on Project-KB to benefit from a higher number of vulnerabilities.

Eclipse Steady is a vulnerability scanner that uses static and dynamic analysis to assess reachability of vulnerable sections of code[43]. As we will see in more detail in other chapters, the two tools are closely related to each other, because Eclipse Steady uses Project-KB as source of data about vulnerabilities, and uses the commit data to identify the vulnerable sections of code. The expansion of the data in Project-Kb would allow Eclipse Steady to detect many vulnerabilities that are currently not covered by the project.

However, Eclipse Steady takes a relatively long time to process the vulnerability data imported from Project-Kb, and the expansion of the database would increase this time even more. The company required changes to be implemented in Eclipse Steady to reduce the execution time and improve its maintainability and architecture.

1.3 Objectives

The present study has 3 main objectives which are explained in detail in the following subsections.

1.3.1 Overview of the Current State of Vulnerability Databases

Many vulnerability databases have been formulated with different purposes, have characteristics in common and contain overlapping information. An objective analysis can help software developers and security researchers decide which datasets are more appropriate for their specific needs. Besides that, this analysis also helps us understand how these databases can be used to expand Project-KB.

For this reason the first goal of this study is to provide an overview of currently available vulnerability databases, its different features and possible applications in software security. We also provide a set of measures that evaluate the consistency between pairs of databases, and use this in association with the knowledge of the databases formulation processes to analyse the reliability of each one of them.

1.3.2 Expanding Project-Kb

We developed a pipeline to adapt the entries in some databases to the format used by Project-KB, allowing for its expansion in the future. The result is larger database

1.3.3 Improving Eclipse Steady

The final goal is to improve the efficiency of some processes in Eclipse Steady, allowing it to import vulnerabilities in less time.

1.4 Document Structure

In chapter 2 we introduce some concepts that will be useful to understand other parts of the study. More specifically, we will briefly explain the Security Content Automation Protocol (SCAP) and Eclipse Steady.

In chapter 3 we present an overview other studies with closely related goals. That is on one side studies that compare and evaluate vulnerability databases and in the other side studies that aggregate data from different sources to build new vulnerability databases.

In chapter 4 we present an overview of the current state of vulnerability databases, with a focus on databases that cover open source software and code-level data.

In chapter 5 we describe the implementation of the pipeline we used to achieve the two first goals described in section 1.3 and the improvements done in Eclipse Steady to achieve the last goal.

In chapter 6 we present the results obtained using the pipeline. We present a detailed quantitative analysis on the existing databases and finish the chapter by briefly describing the database obtained by creating statements in the Project-KB format.

Finally, in chapter 7 we present our final conclusions and possibilities for future work.

Chapter 2

Background

In this chapter we explain concepts that will be relevant for understanding other parts of this study.

2.1 Security Content Automation Protocol

Security Content Automation Protocol (SCAP) is a set of standards for automated vulnerability management that enables interoperability between tools [26]. The standards include a series of catalogs and systems for cataloging and assessing severity of vulnerabilities. Of particular importance for the present work are the Common Vulnerabilities and Exposures (CVE), Common Weakness Enumeration (CWE), Common Vulnerability Scoring System (CVSS) and Common Platform Enumeration (CPE).

In this section we explain these standards, as they will be relevant for understanding the content of many vulnerability databases.

2.1.1 Common Vulnerabilities and Exposures

The CVE system provides a list of records for publicly disclosed vulnerabilities. It is a set of entries for specific instances of vulnerabilities within software projects. Each entry in this system includes an identification number, description and one or more public references. This system is operated by the Mitre corporation and maintained through the collaboration of many institutions, recognized as CVE Numbering Authorities (CNA).

2.1.2 Common Weakness Enumeration

For identifying the type of the vulnerability, the CWE system is used. This is a category system also maintained by the Mitre corporation, and is used to identify and catalog common types of vulnerabilities found in software systems, independently of its instantiations in specific projects.

For example, CWE-787 refers to “Out-of-bounds Write” and CWE-79 refers to “Improper Neutralization of Input During Web Page Generation (’Cross-site Scripting’)”

2.1.3 Common Vulnerability Scoring System

CVSS is a standard for assessing the severity of vulnerabilities.

It defines a set of base metrics:

- Attack Vector
- Attack Complexity
- Privileges Required
- User Interaction
- Scope
- Confidentiality Impact
- Integrity Impact
- Availability Impact

These scores are to be assigned through the analysis of an specialist following the specifications of the system [12]. They are then combined to yield a result from 0 to 10. Optional metrics called *Temporal Metrics* and *Environmental Metrics* can also be assigned to refine the evaluation.

2.1.4 Common Platform Enumeration

Standard for assigning names to IT products and platforms. The CPE identifier is divided in the following parts:

```
cpe:<cpe_version>:<part>:<vendor>:<product>:<version>:<update>
:<edition>:<language>:<sw_edition>:<target_sw>:<target_hw>:<other>
```

The component `part` indicates whether the product is hardware (`h`), application (`a`) or operating system (`o`). This information will be useful later in this study, since the hardware vulnerabilities are not relevant for us and can be filtered.

2.2 Eclipse Steady

As previously mentioned, code-level vulnerability data can be useful to increase accuracy of vulnerability scanners. Some vulnerability scanners, like npm audit [27], do not use this kind of data and are only able to determine the presence of vulnerable dependencies, without verifying the reachability of the vulnerable code. This can lead to false positives since in many cases the vulnerable functions of the libraries are never used in the context of the application that imports it.

False positives are a significant problem for software developers since there are costs in treating potential threats and it is important to know what should be prioritized.

A study published in 2015 [29] demonstrated the use of dynamic analysis for determining reachability of vulnerable code. This paper explores the use of known patches for the vulnerabilities and uses them to determine what is the vulnerable part of the source code. This approach was used to develop the Vulas tool and later used to develop an Eclipse plugin called Eclipse Steady.

The architecture implemented includes a Patch Analyzer, whose function is to interact with a versioning system to compare the constructs in the source code before and after the commits that fix the vulnerability. The constructs changed by the commits are considered to be vulnerable.

A Runtime Tracer is responsible for collecting the trace for each executed programming construct. Then, it is possible to verify if any of the executed constructs are contained in the set of vulnerable constructs.

This approach is not able to completely avoid false positives and false negatives. False positives can be reported because vulnerability exploitability may depend on factors such as the presence of sanitization techniques, which are not analyzed in this approach. False negatives may happen mainly for insufficient coverage by the dynamic analysis.

The problem of false negatives can be mitigated by adding well written tests that ensure good coverage, but it is not possible to ensure that all dangerous paths are analysed. Often a few corner cases can be of particular importance to the attackers.

The article concludes that this approach using dynamic analysis provides strong evidence that a vulnerability is reachable in the context of an application, but not as much strong evidence in the complementary case, when the vulnerability is unreachable. In the latter case, static analysis can more safely assure that the constructs are unreachable.

Considering this point, the research was further developed in [31] adding the use of static analysis. A call graph is built allowing to search for execution paths to the vulnerable constructs. The call graph however is only an approximation and the static analysis struggles to deal with some Java features such as reflection, and service loaders. This difficulty can lead to false negatives, but in many cases can be overcome by the dynamic analysis.

The researchers point that static and dynamic analysis act complementary to each other, because each one has different limitations. In this new version of the tool, it is possible to start the static analysis from methods executed by the dynamic analysis. This feature allows to expand the set of found constructs, often reaching vulnerabilities that would not be found by any of the two types of analysis alone.

Chapter 3

Related Work

In this chapter we present other studies aimed at comparing or aggregating information from Vulnerability Databases.

3.1 Studies Analysing and Comparing VDB's

Other studies presented overviews of vulnerabilities databases analysing its data quality, features or applications. In the following subsections we present some of these studies.

3.1.1 Taxonomic Analysis of Classification Schemes in Vulnerability Databases

[48] provides a comparative study of information contents in prominent vulnerability information sources. The study compares the features and information present in IBM ISS X-Force, NVD, Security Focus, OSVDB, Secunia, US-CERT, VNDB and CVE. The focus of this study is on classification schemes used by different databases to determine cause, impact and severity of vulnerabilities. The authors provide an overview of the features and classification schemes used in each of these databases, and argues for the necessity of a common classification scheme.

3.1.2 A Survey on Vulnerability Assessment Tools and Databases

The study in [21] presents an analysis on open-source databases with the purpose of enabling developers to make informed decision on which ones to select. The authors determine a set of criteria to be taken into consideration when making this decision.

The first set of criteria is the information covered by the databases regarding the vulnerabilities. Ideally, each entry of the database should include information about the scope, impact & risk, resolution, affected products and their vendors, exploit, categorisation and relations with other vulnerabilities.

The second set of criteria is based to the main capabilities of the VDB in terms of interfacing, standards support, information freshness and user support.

Twenty databases are considered in the study, including not only databases for instances of vulnerabilities (e.g. CVE and NVD), but also other kinds of data, like classification schemes (e.g. CWE) and rankings (e.g. OWASP Top 10). An overview is provided on the evaluation of these databases according to the mentioned criteria.

3.1.3 Which is the Right Source for Vulnerability Studies? An Empirical Analysis on Mozilla Firefox

This study presents an analysis of the impact of data quality on vulnerability studies. It presents an overview of features of different databases and their applications to other studies. A total of 14 databases are included.

The study presents an analysis on the data quality of Mozilla Firefox Security Advisories, and concludes that many papers based on it potentially have compromised results because the data is incomplete and biased, partially due to the fact that this database only includes fixed vulnerabilities.

3.2 Aggregation and Crawling of VDB's

We found four other works that, like ours, aim at extracting information from one or more VDB's to obtain a database with code level data. The database most commonly used for this is NVD.

Each study presents its own methods to extract data from it and small differences in implementation lead to different results even when using the same data sources. For example they may consider different repository domains, such as Github, Google Android or Mercurial.

In the following subsections we provide a brief description of each these studies and their methods.

3.2.1 FASTEN Vulnerability Producer

The goal of the FASTEN project is to use software package management more intelligent to make software ecosystems more robust [10]. The project relies on the creation of Fine-Grained Call Graphs (FGCG) at function level to assess risk posed by vulnerable code.

In order to create the FGCG's it needs to gather information from vulnerability databases containing code level data. For this purpose the FASTEN Vulnerability Producer was developed. It aggregates and enriches information from the following sources:

- NVD JSON Feed
- GitHub Advisories
- MSR 20191

- MSR 20202 (Big-Vul)
- Safety DB
- cvedb (by fabric8-analytics)
- victims-cve-db
- Debian Security Tracker
- SAP project-kb

For this purpose, a parser was created for each these databases.

The references that are found to be associated to each vulnerability are handled by the program to figure out whether it is possible to use them to extract *patch diffs*, that is the lines of code changed from the vulnerable version to the patched version of the project. The following sources are considered in order to obtain this information:

- GitHub Commits
- GitHub Pull Requests
- GitHub Issues
- GitLab Commits
- GitLab Merge Requests
- GitLab Issues
- BitBucket Commits
- BitBucket Pull Requests
- BitBucket Issues
- Bugzilla bugs
- JIRA tickets
- Git Trackers Commits
- SVN Revisions
- Mercurial Revisions
- Apache Mailing List

3.2.2 Big-Vul

Obtained crawling from NVD and looking for paths to commits [9]. CVE entries that have references to repository paths are selected and the crawler looks for traversal paths to code commits. The work was focused on Github repositories and a few products with their own Git server, such as Google Android. It looks for additional information in `https://www.cvedetails.com/`.

The entries include the following features: CVE identifier, CWE, description, publish and update dates, list of commit links, changed files and metrics associated to the vulnerability severity and impact.

3.2.3 CrossVul

Similarly to Big-Vul, this dataset was built by crawling the NVD for Github repository links [24]. The references to commits found in the entries are kept as they are and references to Pull Requests are used to collect sets of commits. They also filter invalid links and links that do not point to pull requests or commits. The files modified in the commits are retrieved using the git-diff command and the test files are filtered.

This dataset contains CVE identifiers, CWE, commit links and changed files.

3.2.4 Patchworking

The goal of the study is to analyse vulnerability patches according to its classification and verify if vulnerabilities of the same type are solved by applying the same kind of code transformations [4].

For achieving this goal, the authors used entries from Project-KB and expanded the dataset by mining NVD. They used NVD entries from 2002 to 2018 and filtered out entries already included in Project-KB. To obtain links to repositories and fix commits they extracted URL's containing the substrings `git`, `svn`, `cvs`, `hg` or `mercurial`, and reverse-engineered them to obtain commit ids. Their analysis was focused on Java projects, so they removed URL's that pointed to repositories that did not contain any Java files.

This dataset includes fields for repository links, commit identifiers, CVE's, CWE's, CWE descriptions and repository type (which can be git, svn, mercurial or cvs).

3.3 Conclusion

In relation to the other studies that analyse and compare VDB's, we present the following new contributions:

- We focus on databases that are useful for open source security and analyse the presence of code-level information in each database.

- We analyse a different set of databases, including some that have been developed more recently and databases developed for academic studies.

- We present an update on the state of the main Vulnerability Databases. Some of the mentioned studies were performed years ago and contain outdated information.
- We provide a more detailed quantitative analysis which was not present in these studies, measuring languages of the affected projects, overlaps between databases, and other useful information.
- We present a method to measure the completeness and consistency of each database with relation to the others, as well as the conclusions obtained by measuring it.

Compared to Big-Vul, CrossVul and Patchworking, our study presents similar methods for extracting information from NVD, but also aggregates information from multiple other sources, and our implementation is easily extensible to include even more sources.

FASTEN is the work whose goals are most similar to ours, since it also aims at aggregating vulnerability data from several sources with the final objective of enabling code-level analysis. Our project covers a different set of databases and the final format obtained is different because it is specifically targeted to Project-KB.

Chapter 4

Databases for Publicly Known Vulnerabilities

In this chapter we provide several examples of databases of disclosed open source vulnerabilities. Some of the datasets are obtained by automatic processes, others are manually curated. Different datasets contain different kinds of metadata related to such vulnerabilities and allow vulnerability scanners to implement different approaches to its assessment and mitigation.

We analyze the purposes, sources, formulation process and features of each of them. Particularly interesting for our study is whether the databases include fields for representing fix commits in a structured way. This allows us to more easily and reliably adapt the data to the format used in Project-KB.

4.1 Project KB

Project-KB was created at SAP with the main purpose of feeding the vulnerability scanner Eclipse Steady, but also to provide data to enable research in security [30].

The main source for the data in Project-KB was the NVD particularly focusing on Java projects used at SAP products or internal tools.

The project is open source and is maintained by SAP security researchers.

Includes for each vulnerability the corresponding CVE or other identifier. Can contain a list of fix commits or a list of affected and unaffected artifacts or both.

4.2 National Vulnerability Database

NVD is the U.S. government repository of vulnerability data. Content of NVD is based on analysis of the entries published to the CVE Dictionary. NVD relies on vendors, third party security researchers and vulnerability coordinators to provide information about vulnerabilities which are then analyzed by NVD staff and used to associate more data to each CVE using SCAP.

Each entry contains data about one vulnerability, identified by one CVE, to which a CVSS score and one more CWE's and CPE's may be associated. A text description is also provided for each vulnerability.

NVD does not contain any field specifically for listing code-level information, like fix commits. However, the entries usually contain also URL's to related resources. This resources may include advisories pages, software repositories, fix commits, pull requests issues, etc. NVD does not provide any guarantees regarding these references. As stated in their website: "No inferences should be drawn on account of other sites being referenced, or not, from this page."

4.3 OSV

Created by Google with the main goal of providing information about when security vulnerabilities were introduced and fixed. Provides a common schema for representing Open Source Vulnerabilty data. The schema provided by OSV is one of the most complete in terms of features provided. Besides the data included by NVD, the OSV also provides fields for affected packages, the ranges of affected versions, commits that introduced and fixed the vulnerabilities, list of aliases (for vulnerabilities that have different id's in different databases) and credits field, which specifies the person or entity responsible for the discovery, patch or other event in the vulnerability life cycle. However, these fields are not necessarily present for all the vulnerabilities. Also contains a field for severity score, which can be given using CVSS. The schema not support other metrics but may be extended for this purpose.

OSV database includes vulnerability data aggregated from different sources:

- Github Advisory Database
- Python Packaging Advisory Database
- Go Vulnerability Database
- Rust Advisory Database
- OSS-Fuzz
- Global Security Database (GSD)

In the following sections we see each of them in more detail.

4.3.1 Github Advisory Database

Public database maintained by Github [**?**]. Any person with admin permissions in a Github repository can create an advisory and request Github to review it. Github does not explain in detail how the review process works.

The database contains more than 170,000 vulnerability reports, of which more than 9,000 were reviewed [15]. The reviewed entries normally include references to affected and patched versions. There is no field for fix commits, but sometimes contains reference links to them.

4.3.2 Python Packaging Advisory Database

Entries are created or edited by creating Pull Requests in a Github repository [33]. Many of the existing entries were discovered automatically using a tool called Vuln feeds. This tool uses a set of heuristics to match CVE's with corresponding packages and versions. It can yield false positives, and sometimes human action is required to decide on how to proceed.

The repository belongs to Python Packaging Authority (PyPA) It is a working group that maintains a core set of software projects used in Python packaging [33]. PyPA goals include: "To provide a relatively easy to use software distribution infrastructure that is also fast, reliable and reasonably secure." [34]

4.3.3 Go Vulnerability Database

Vulnerabilities reports are curated by the Go Security team. The data comes directly from Go package maintainers or sources such as MITRE and GitHub [16].

4.3.4 OSS-Fuzz Vulnerability Database

OSS-Fuzz is a fuzzing tool developed by Google [17]. Fuzzing is a kind of testing technique that consists in providing many different inputs to a program and monitoring it for unexpected behaviors.

This tool is continually used for testing open source projects and its findings are published in an open repository in Github.

The analysis performed by the tool allows to determine the commit ranges and versions in which the vulnerabilities affect the project. The resulting data is then automatically imported to the repository, without reviews by humans. The entries are regularly updated by an automated process, but can also be manually updated by users, via Pull Requests.

4.3.5 Rust Security Advisory Database

Maintained by the Rust Secure Code Working Group based on security advisories filed against Rust crates published in `https://crates.io` [41]. Security bugs are reported in pull requests in an open repository.

The repository is available in markdown format, with a set of fields that are specific to this database, and also in json format, following the OSV schema.

Sometimes the entries include a list of affected functions, which is an information that is normally not available in other databases that use the OSV format.

4.3.6 Global Security Database

Created by the Cloud Security Alliance, a nonprofit organization, with the goal of addressing gaps between the vulnerability identifiers space and the current needs of industry [2]. The project follows an open source model.

4.4 Ruby Security Advisory Database

Open source database maintained by the users community in a Github repository [39]. New entries are introduced by users via Pull Requests.

There is a script created for synchronizing this with Github Advisory Database, verifying if there are vulnerabilities that were still not commit to the Ruby database. However this process is not completely automated, and this script requires a user to complete the vulnerability data afterwards.

The database does not include fix commits, but includes lists of patched and unaffected versions.

4.5 SECBENCH

Database created mainly for the purpose of testing static analysis tools [38]. Results from mining 248 projects from Github for 16 different patterns of security vulnerabilities and attacks.

The repositories were selected in a way to ensure the presence of different programming languages, principally the 5 most popular languages in Github at this time: JavaScript, Java, Python, Ruby and PHP. The authors also ensured the presence of repositories of different sizes. Their goal by doing this is not to ensure a good coverage of the most relevant projects, but rather to have a dataset that is representative of the whole Github. That involves the inclusion of projects of different sizes, which are maintained by developers with different skill levels.

The mining was done in two steps, first looking for a set of patterns in commit messages and then manually verifying the commits, including the source code. Most vulnerabilities do not have CVE's or any other identifier.

4.6 Devign

Manually collected dataset used to train a machine learning model [53]. Collected from only 4 projects: FFmpeg, Qemu, Linux, Wireshark [52].

Their process was composed of two main steps: first the commits are checked by a group of bachelor security researchers and labels are assigned to it, then, if there is an inconsistency between them, a senior researcher verifies it. If the group is not sure about a commit, it remains out of the dataset.

The dataset includes fix commits and code of the affected functions for all the entries. Each entry contains exactly one fix commit. It does not contain any vulnerability Identifiers, CWE, CVSS or any other metrics.

4.7 Commercial Databases

Some companies develop their own vulnerability databases for the purpose of feeding their vulnerability scanners. Each of these is collected in a different way and contains features that are appropriate to the needs of their products. The databases presented here are not necessarily in public domain, but their data is accessible in the respective websites or API's.

4.7.1 Snyk Vulnerability Database

Maintained by Snyk Intelligence Security based on multiple sources. The database is publicly available and contains 1166 vulnerabilities disclosed by Snyk Security Researchers [46].

Like NVD, Skyk database provides CWE, CVSS and a list of reference links. Not all the vulnerabilities in the database have an associated CVE, because the data comes from multiple sources. It also provides information about affected packages and versions, but not using CPE. Each entry contains a text field describing how to fix it, which in many cases indicates the patched version to which the affected package can be updated.

Besides the CVSS, it also includes scores to attack complexity, and potential damages to confidentiality, availability and integrity of the system [46]. This scores helps the developer evaluate the severity.

All the entries also include a credits field for the entity that disclosed the vulnerability.

4.7.2 White Source Vulnerability Database

Maintained by White Source by aggregating data from multiple sources [50]. Does not contain references to commits in a structured format, but has a text field for fixes. This field often contain a reference to a fixed release. Also provides CVE, CWE and CVSS. Does not contain any field specifically for listing affected versions.

4.7.3 Debricked

Debricked collects vulnerability data from NVD Database, NPM, C# Announcement, FriendsOfPHP's security advisories, Go Vulnerability Database, PyPA Python Advisory Database, GitHub Issues, GitHub Security Advisory, and others [7]. The process is fully automated, and involves steps to clean the data, but the methods are not explained in the documentation.

The entries include identifiers, CVSS scores and lists of affected and safe dependencies [8]. Also contains a field with suggested fixes, indicating the lines of code that need to be changed. However all this information is not necessarily available for every vulnerability.

Database	**Fix Commits**	**Affected Versions**	**Patched Versions**	**CVE**	**CWE**	**Assessment Metrics**
NVD	N	Y	N	Y	Y	CVSS
Project-KB	Y	Y	Y	Y	N	N
OSV	Y	Y	Y	Y	Y	CVSS
Ruby	N	Y	Y	Y	N	CVSS
SECBENCH	Y	N	N	Y	Y	CVSS
Devign	Y	N	N	N	N	N
Big-Vul	Y	N	N	Y	Y	Several
CrossVul	Y	N	N	Y	Y	N
Patchworking	Y	N	N	Y	Y	N
Snyk	Y	Y	Y	Y	Y	Several
Whitesource	Y	N	Y	Y	Y	CVSS
Debricked	Y	Y	Y	Y	Y	CVSS
Safety DB	N	Y	N	Y	N	N

Table 4.1: Databases features

4.7.4 PyUp Safety DB

Maintained by PyUp Cyberintelligence Team which monitors Python packages for security vulnerabilities [37]. These database is used to feed their vulnerability scanner [36].

The entries include CVE's or identifiers following their own enumeration system, a description of each vulnerability and an indication of what are the affected versions. There are no fields for repositories or fix commits, but this information can often be found in the description.

This database is under the license *CC BY-NC-SA 4.0* which allows for copy, redistribution and transformation of the material under certain conditions for non commercial purposes.

4.8 Conclusion

We list in table 4.1 what features are covered by the format provided by each vulnerability database. We only analyse here whether the feature can be present in a structured way, meaning that there is a field dedicated to this information in the database. Notice that the format used by OSV is also implemented by other databases (PyPA, Go, RustSec, Github, GSD), so they can have the same features.

The databases that include fix commit data in a structured way are specially interesting for the present study, since they can be easily adapted to Project-kb's format. Databases which do not include a field for this information can still be used for this purpose because the fix commit can in some cases be extracted from other fields (e.g. the description).

We noticed there are many database whose formulation process include an automated importation of entries from other databases. To help understand the sources of data of each one of them we show these relations in figure 4.1. In this diagram we represent these relations using arrows pointing from data sources to the databases to which they are imported. This figure illustrates only

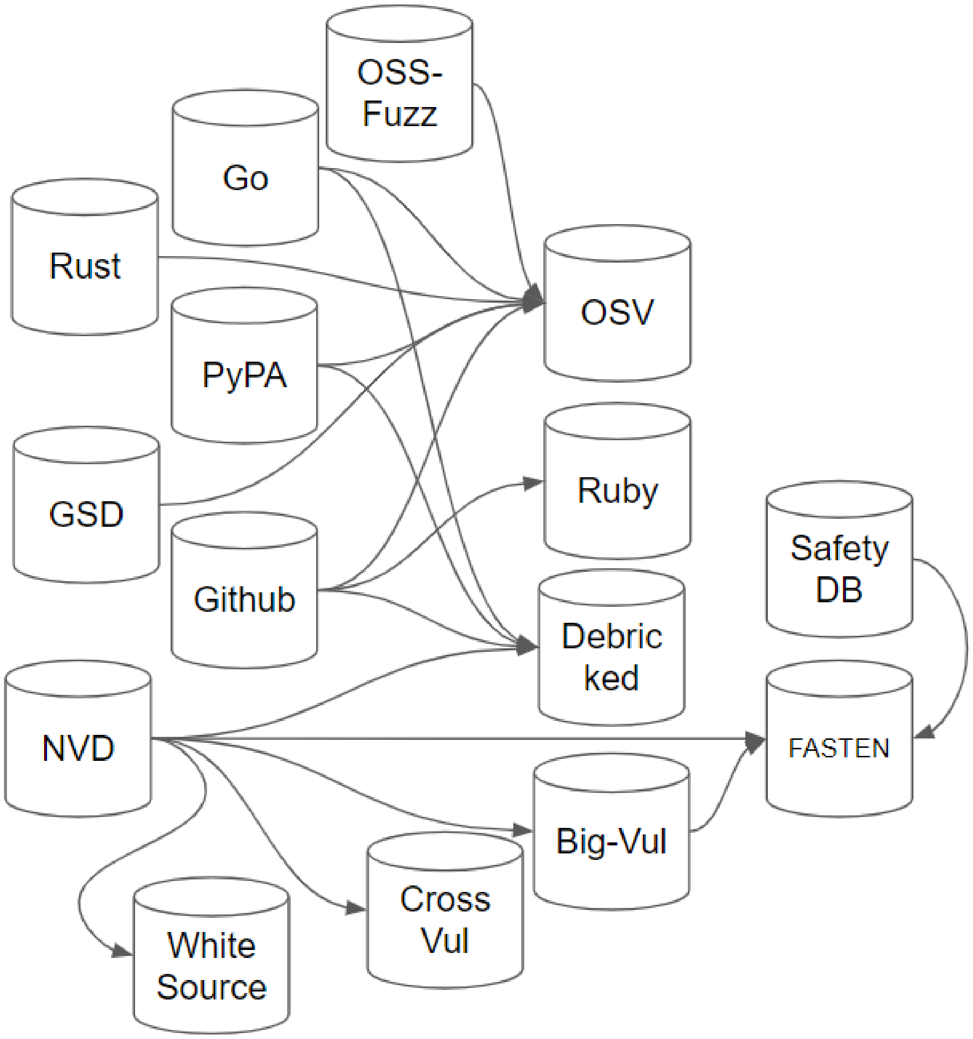


Figure 4.1: Sources of Data

the existence of automated processes integrating the databases and is not necessarily complete, some sources are left out of this diagram. Notice also that the existence of an automated process connecting two databases does not necessarily imply that all of the entries of one of them are imported to the other. In this diagram are not included the databases that don't have any connection of this kind to the other databases.

Chapter 5

Implementation

Some of our main goals are the quantitative analysis of a set of Vulnerability Databases, extraction of code-level data from them and the creation of statements in Project-KB's format.

These goals are achieved by the same program, which is described in detail in this chapter.

5.1 Pipeline

In figure 5.1 we see the pipeline created to achieve our goals. The steps in this process are the following:

1. Import data from the different databases
2. Extract statistics from the objects obtained from the Import step.
3. Overlap between each pair of databases is measured and organized in a matrix.
4. Consistency between datasets is measured
5. Merge all the databases in a single one
6. Filter vulnerabilities that can not be used by Project-KB, either because they lack information or because they do not affect open source software.
7. Export statements in the format used by Project-KB

The last 2 steps are performed both for all VDB's separately and for the merged VDB.

In the next section we explain in detail all these steps as well as the program architecture.

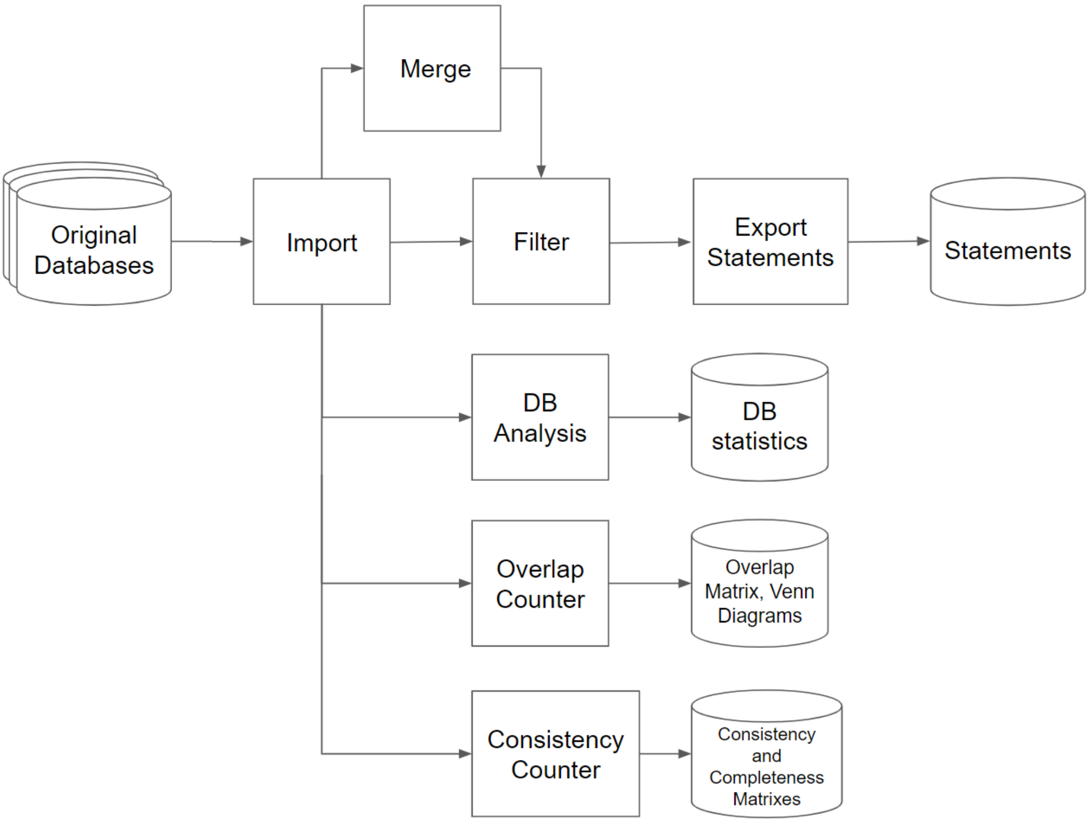


Figure 5.1: Pipeline

5.1.1 Import Databases

Each of the presented vulnerability databases stores its data in a different format. For this reason it was necessary to develop scripts for extracting the necessary data from each of them. The architecture used in the implementation includes an abstract superclass determining the interface for importing the databases and one subclass for each database format.

This architecture is intended to facilitate the extension of the program to include new databases. That also allows us to store all the required data in a unified format, defined by the classes *Vdb* and *Vulnerability*. We see in figure 5.2 a UML diagram representing the architecture for the factories responsible for importing the vulnerabilities.

5.1.2 DB Analysis, Overlap and Consistency

This part of the pipeline is used to characterize the data from the different databases. The methods and results are explained in detail in chapter 6.

5.1.3 Filter

There are two criteria for excluding vulnerabilities:

- We filter out vulnerabilities that only affect hardware components, because our work is targeted on software vulnerabilities

- We filter out vulnerabilities that do not have enough information to generate a statement. Each vulnerability needs the URL to a repository and at least one commit id

5.1.4 Merge

All the data from the VDB's is put together in a single data structure. When a vulnerability is present in more than one database, the information of all the entries is merged. This means the resulting entry will have all the aliases (identifiers) and commits from the previous ones.

In case two entries corresponding to the same vulnerability have different sets of commits, the union between these sets is calculated to form the new entry. When applied to the reachability analysis in a vulnerability scanner, this approach means a higher number of constructs or lines of code will be flagged as vulnerable, therefore decreasing the chance of a false negative but increasing the chances of a false positive in case one the VDB's actually had inaccurate data. We prefer to follow this approach because false negatives could present risks to a project, letting a vulnerability pass undetected, while false positives may be inconvenient for the developers but would would not cause security problems.

5.1.5 Export Statements

After filtering and merging the databases we proceed to the export step. The goal of this step is the creation of YAML files with the fields required by Project-KB. A complete statement will have the

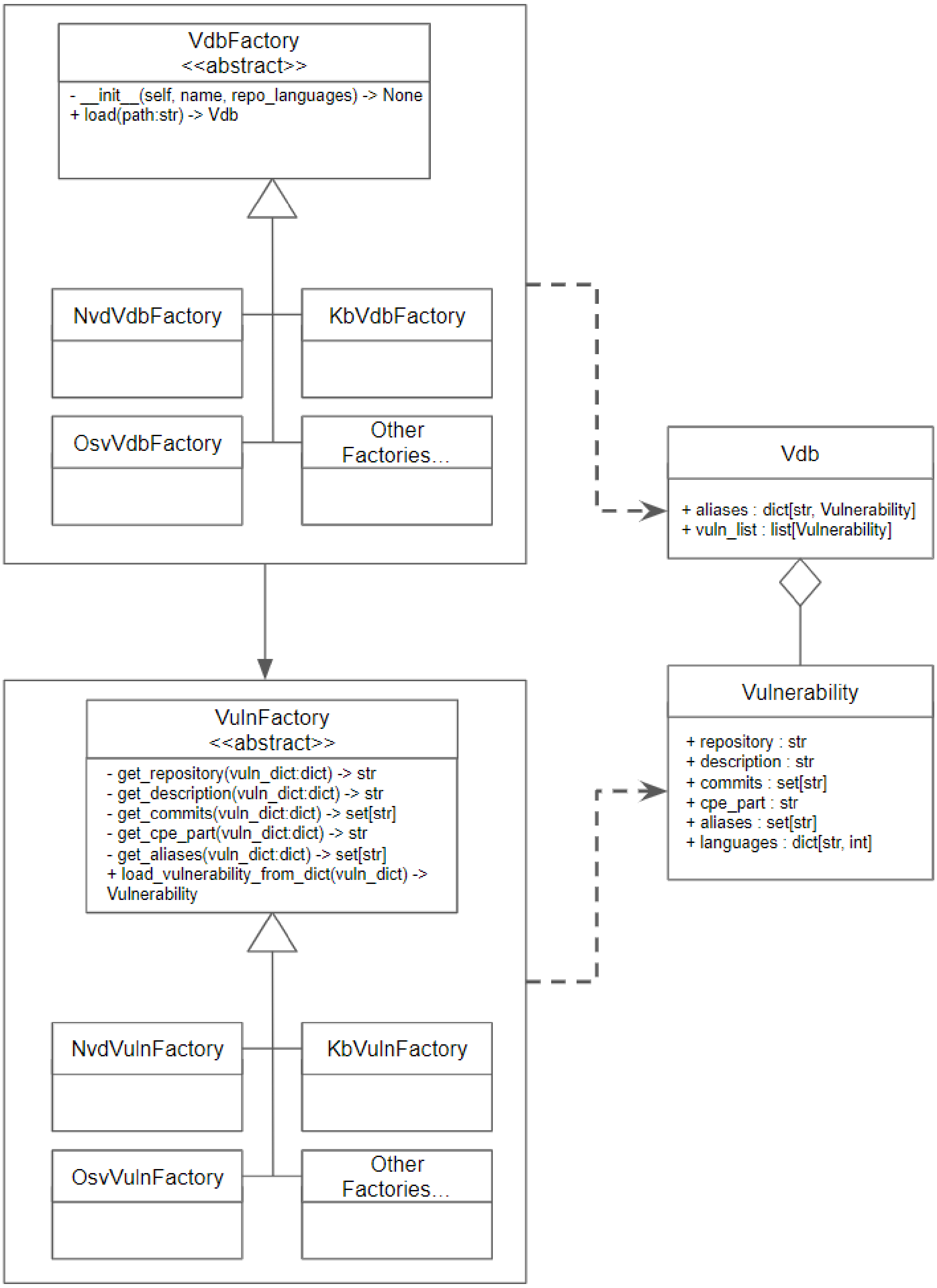
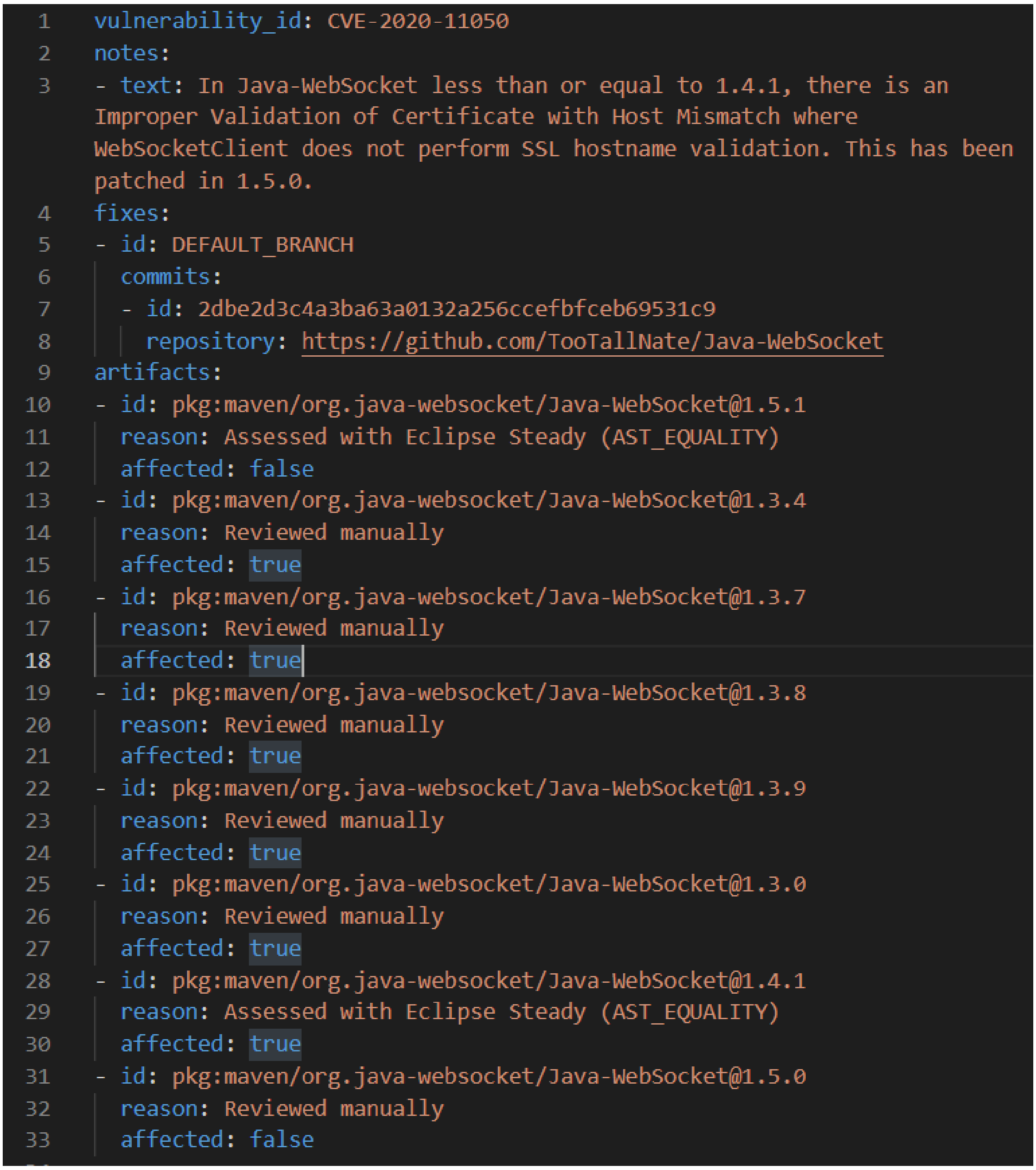

VdbFactory
<<abstract>>
- __init__(self, name, repo_languages) -> None
+ load(path:str) -> Vdb
NvdVdbFactory
KbVdbFactory
OsvVdbFactory
Other Factories...
Vdb
+ aliases : dict[str, Vulnerability]
+ vuln_list : list[Vulnerability]
Vulnerability
+ repository : str
+ description : str
+ commits : set[str]
+ cpe_part : str
+ aliases : set[str]
+ languages : dict[str, int]
VulnFactory
<<abstract>>
- get_repository(vuln_dict:dict) -> str
- get_description(vuln_dict:dict) -> str
- get_commits(vuln_dict:dict) -> set[str]
- get_cpe_part(vuln_dict:dict) -> str
- get_aliases(vuln_dict:dict) -> set[str]
+ load_vulnerability_from_dict(vuln_dict) -> Vulnerability
NvdVulnFactory
KbVulnFactory
OsvVulnFactory
Other Factories...

Figure 5.2: Factories UML Diagram



```
vulnerability_id: CVE-2020-11050
notes:
- text: In Java-WebSocket less than or equal to 1.4.1, there is an Improper Validation of Certificate with Host Mismatch where WebSocketClient does not perform SSL hostname validation. This has been patched in 1.5.0.
fixes:
- id: DEFAULT_BRANCH
  commits:
  - id: 2dbe2d3c4a3ba63a0132a256ccefbfceb69531c9
    repository: https://github.com/TooTallNate/Java-WebSocket
artifacts:
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.5.1
  reason: Assessed with Eclipse Steady (AST_EQUALITY)
  affected: false
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.3.4
  reason: Reviewed manually
  affected: true
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.3.7
  reason: Reviewed manually
  affected: true
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.3.8
  reason: Reviewed manually
  affected: true
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.3.9
  reason: Reviewed manually
  affected: true
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.3.0
  reason: Reviewed manually
  affected: true
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.4.1
  reason: Assessed with Eclipse Steady (AST_EQUALITY)
  affected: true
- id: pkg:maven/org.java-websocket/Java-WebSocket@1.5.0
  reason: Reviewed manually
  affected: false
```

Figure 5.3: Project-KB Statement File

vulnerability identifier, a description, a list of fixes containing the repository, branch and commit hashes, and a list of affected artifacts. For Project-KB's format, it is not necessary to have both the commits and affected artifacts, but at least one of them. In figure 5.3 we see an example of file in this format. However, we decided to focus exclusively on the exportation of statements with fix commits, because it enables code-level analysis such as the one described in section 2.2.

This step is performed for both the separated databases and the merged one.

By applying it to each VDB separately, we obtain different sets of statements coming from different data sources and the developers can choose which one to use according to their needs.

A developer can for example choose to use only databases that he considers to be more reliable, or only the databases that potentially contain vulnerabilities that affect components used in his project.

5.2 Improvements in Eclipse Steady

When installed for the first time, Steady imports vulnerabilities from Project-KB and performs an analysis that determines the language constructs affected by each vulnerability. The results of this analysis are then uploaded to the backend of the tool and are periodically updated. This process may take 2 to 3 hours to import, analyse and upload 729 vulnerabilities. It is desirable to reduce this execution time, principally considering the fact that, as a result of this study, more vulnerabilities can be imported and the process would take even more time.

In this chapter we describe the improvements made in Eclipse Steady to reduce this processing time and the modifications in architecture to improve its maintainability.

5.2.1 Previous Architecture

Kb-importer is the module of Eclipse Steady responsible for importing and processing data from Project-KB. Here we focus on the architecture of this module, because this is the component responsible for the processes that have to be optimized.

When installing Eclipse Steady, a Docker container is created for kb-importer and a few files are downloaded inside this container:

- `kaybee` — Binary file from Project-KB
- `kaybeeconf.yaml` — Configuration used by kaybee
- `kb-importer.jar` — Jar file for the

The `kaybee` binary reads the configuration specified in `kaybeeconf.yaml` and use it to determine what sources to use to get vulnerability metadata. After pulling this data, it will generate a shell script file containing commands to be executed for each vulnerability. This file contains commands for the following tasks:

1. Cloning git repositories containing the vulnerabilities.
2. Using `git diff` to get the modifications from the fix commits and create files containing them.
3. Call for each vulnerability a command to use `kb-importer.jar` to import it.

Then the part of the program written in Java is responsible for:

1. Sending the results of the `git diff` command to the component patch-lib-analyzer to extract the changed constructs.

2. Sending the final results of this analysis to the backend.

3. Send a list of affected artifacts to the backend.

5.2.2 Profiling

The first step in this work was to use profiling to measure the bottlenecks of the program.

We found that a large percentage of the processing time was used to send requests between kb-importer and two other containers: the backend and rest-lib-utils. These requests were not asynchronous and took around 60% of the time.

We found that there are two `for` loops in which these requests are sent. One of them is the a loop that gets information about libraries affected by the vulnerabilities and uploads lists of affected artifacts to the backend. The other loop is used to analyze contruct changes from the commits. These loops take respectively around 60 %, and 20 % of the processing time.

Also, the fact that the Java program was called once for each vulnerability made it necessary to initialize the JVM multiple times. We considered the possibility that this could be taking a significant time, but the profiling results showed that only a very small percentage of the time was used in this process (less than 1%).

Based on this information, we concluded the performance could be improved by reducing the number of requests and processing vulnerabilities in parallel.

5.2.3 New Architecture

The improvements to performance were achieved with two main modifications:

The first modification is to check if a vulnerability is already present in the backend before importing it. If the vulnerability is already present, it is not necessary to perform any analysis and no more requests are sent. However, we also decided to provide the user with the option to overwrite the vulnerabilities in the backend, in case it is necessary to modify the data for any reason.

The second modification is the parallelization of the processes in kb-importer. Each vulnerability can be imported in a separate thread. The repositories required for analyzing the fix commits now can be cloned and accessed in parallel. Locks were created to avoid simultaneous access to the same cloned repositories, because it could cause conflicts.

Besides these modifications, we also modified the following aspects to make the software easier to debug and maintain:

- We implemented a REST API in kb-importer. This allows the user to send requests to initiate or stop processes and to check the current state of the program. This also makes the architecture of kb-importer more similar to that of other components of the project.

- All the processes previously implemented in shell script were passed to Java.

Configuration	Previous Architecture	New Architecture
1	N/A	10s
2	2h	25min
3	3h	-

Table 5.1: Time to Process Vulnerabilities

As a consequence of this modifications, the program only needs to be called once to process all the vulnerabilities. Execution options previously specified by environment variables or as arguments to the Java program are now specified as arguments in the requests to the API.

5.2.4 Final Results

We measured the total execution time for a set of N vulnerabilities using different configurations:

1. Using the option *overwite=false* and having all the vulnerabilities already in the backend. That means the program only has to send requests to the backend to check the presence of the vulnerabilities, but will not have to analyze and upload them.

2. Using the options *overwite=true* and *skipClone=true*. All the vulnerabilities have to be analyzed and uploaded to the backend, except for the ones that would require cloning a repository.

3. Using the options *overwite=true* and *skipClone=false*. All the vulnerabilities have to be analyzed and uploaded to the backend and some repositories have to be cloned to allow the execution of *git diff* for some vulnerabilities.

All the experiments were performed in a Ubuntu virtual machine with 8 Gb of RAM and 4 processor cores. The processor used was an Intel Core i5 vPro 8th Gen. Table 5.1 shows the approximate results obtained for each configuration. The first configuration was not relevant in the previous architecture since the program did not check the presence of vulnerabilities in the backend

Chapter 6

Quantitative Analysis on the Existing Databases

In this chapter we present quantitative results obtained from analysis performed on each database. These results help us characterize the data that can be added to Project-KB and can also assist software developers and security specialists with objective information to make a decision on which databases to use.

We measured the size of each database, the distribution of programming languages of the repositories containing their vulnerabilities, number of entries containing repositories and commits, and the overlaps between databases. We will see the results in detail in the next sections. All the results were obtained based on the state of the databases in September 16th, 2022.

Of the databases previously mentioned in chapters 3 and 4, we excluded some of them for the following reasons:

- Snyk, Whitesource and Debricked are commercial databases and that brings potential problems with their licences or availability of the data. Their licenses would not necessarily allow for the use in Project-Kb, and even if they did, they were not available for download or accessible via free API's.

- Devign does not include identifiers associated to the vulnerabilities, and that would complicate some of the analysis done in this study.

- Due to time limitations we could not analyse and write importers to some of the databases referenced in the related work such as Debian Security Tracker.

- Also due to limited time, we decided to not develop an importer specifically for GSD, which was available in a format which is not exactly compatible with the OSV schema. However, since OSV imports data from GSD, the vulnerabilities in GSD are included in the results we present regarding OSV.

- A few databases mentioned in other studies do not exist anymore. That is the case for example for OSVDB, which was shutdown in 2016 [20].

6.1 Repositories

For each Database we developed methods to look for project repositories in the entries. As already described, in some databases the repository is listed in a structured way, with fields reserved for this, while others contain text fields or URL fields that sometimes contain a reference to the repository.

We used the following methods to extract repository links from the entries in the databases:

- **NVD** — We first check the list of references to verify if any of them is a list to a git repository. If no references are found, we check if there are URL's in the description field.
- **OSV** — The field `affected.ranges` is used to indicate ranges of versions affected by the vulnerability. When the type of range is indicated as `GIT`, the range is given in the form of commit hashes and a subfield is used to indicate the repository. However, this field is not present for all the entries. Another field that can contain the repository link is the list of references. If the reference type is indicated as `FIX`, the field contains a URL to the repository.
- **Ruby** — The entries usually contain a `url` field which contains a link to a repository. If this field is not present, or the URL is not valid, we check the URL's listed in the list of references, the same way as for the NVD.
- **Project-KB** — When a vulnerability contains commit information, the repository URL is always present.
- **Secbench and Patchworking** — Both systematically list the repository name or link for all the vulnerabilities
- **Safety DB** — There are no fields for URL's, so we look for it in the vulnerability description in the field `advisory`.

In the cases where the field is not guaranteed to contain a repository URL we perform 3 verifications to filter URL's that are not linking the repository of the vulnerable project:

1. We check if the URL is in a set of known domains. We consider the following domains for the repository of the vulnerable project:
    - github.com
    - gitlab.com
    - git.kernel.org

Database	**No. Entries**	**With Repository**	**github.com**	**git.kernel.org**	**others**
Patchworking	471	471	339	0	132
Secbench	662	662	662	0	0
CrossVul	5131	5131	5131	0	0
Project-KB	729	647	587	0	60
PyPA	1579	1278	1275	0	3
Ruby	639	266	264	0	2
Big-Vul	3977	3977	3977	0	0
OSS-Fuzz	2430	2423	1864	0	559
OSV	27783	19872	8852	10079	941
Github	6622	5517	5496	0	21
SafetyDB	3643	924	920	0	4
RustSec	434	389	381	0	8
NVD	195571	27752	25539	1660	553
Go	251	243	162	0	84
Total	203716	43340	30420	10577	2343

Table 6.1: Vulnerabilities with Repository Links

- googlesource.com
- git.apache.org
- git-wip-us.apache.org

Repositories in other domains are not considered. In any case, we noticed most repositories are contained in this set of domains, mainly in Github.

2. Check if the name of the repository a contains substrings like `CVE` or `Advisory`, because they normally are repositories for exploits or advisories. The following substrings are used to filter the repository names: `cve`, `advisories`, `advisory`, `vulnerability` and `vulnerabilities`. Any repository that contains one of these substrings in its name is excluded. These strings were selected based on the results obtained before adding this step to our process. We manually verified that the results contained a large amount of false positives containing these substrings. The comparison done in this step is case insensitive.

3. If the repository is in Github, check the languages of the repository by using Github's API. If the repository does not contain any language, it is probably not the project that contains the vulnerability (can also be an exploit or advisory). In some cases the URL is broken and we get error 404. These links are also discarded.

In table 6.1 we indicate the number of vulnerabilities in each database to which we were able to identify the corresponding repositories using the methods described above.

Notice that the numbers above may still include some false positives. The rules described above are enough to filter the most common cases of false positives, but do not guarantee that all the repositories found are correctly associated to the vulnerabilities.

6.2 Commits and Releases

As explained, one of the most important aspects for this study is the presence of code-level information in the databases. For this reason, we determined the number of vulnerabilities in each database for which it was possible to extract lists of commits and lists of patched versions.

As seen in the previous chapter, some of the databases and entries contain fields specifically dedicated to this information, while others don't. However, in the cases where there is no specific field for this data, it is often possible to find this information in other fields, such as `references`, `description` or `summary`.

We use the following methods to obtain fix commits from each database:

- **NVD** — We check the references field for commits and pull requests URL's. Then we filter the information the same way as described in the previous section
- **OSV** — In the field `affected.ranges`, when the type is `GIT`, the values of commit hashes are given. In each range there are objects called events and when an event is marked as `limit` or `fixed` it determines the commit after which the software is no longer affected by the vulnerability. We take these commits as fix commits. We also search for fix commits and pull requests in the list of references.
- **Ruby** — We look for commits or pull requests in the `url` and `references` fields.
- **Project-KB, Secbench, Patchworking, Big-Vul and CrossVul** — All of these list the links to commits pull requests or commit hashes systematically in dedicated fields.
- **Safety DB** — The description field often contains a link to a commit or pull request.

As previously explained in chapter 4, some databases also include fields for fixed releases. In these cases, we counted the vulnerabilities with this information.

Table 6.2 shows the number of vulnerabilities to which fix commits and fixed versions were found in each database.

6.3 Vulnerabilities Languages

An interesting information for the developers using the databases is the language of the projects affected by the vulnerabilities. Having this information, the developer can choose to use databases with vulnerabilities that affect projects in the same language used is his own projects. To get this information we separated the vulnerabilities to which Github repositories were found and used Github's API to get the repositories languages. Then we counted the main language for the repository of each vulnerability, i.e. the language with more lines of code.

The chart in figure 6.1 shows the number of vulnerabilities extracted from each database in each of the main languages. For the vulnerabilities that do not have a Github repository associated, we could not determine the languages, so they are listed as `undefined` and represented by the

Database	**No. Entries**	**With Commits**	**With Fix Releases**	**With Commit or Fix Releases**
Patchworking	471	471	0	471
Secbench	662	662	0	662
CrossVul	5131	5131	0	5131
Project-KB	729	647	511	729
PyPA	1579	648	1519	1524
Ruby	639	135	571	574
Big-Vul	3977	3949	0	3949
OSS-Fuzz	2430	2283	0	2283
OSV	27783	16225	13101	25602
Github	6622	2509	5283	5368
SafetyDB	3643	841	3643	3643
RustSec	434	19	298	306
NVD	195571	10863	0	10863
Go	251	243	240	249
Total	200234	24787	3241	27163

Table 6.2: Vulnerabilities with Releases and Commit Data

black bars. In figures 6.2 and 6.3 we show the same chart, but excluding the largest databases to facilitate visualization of the small databases.

The black bars are either explained by the absence of links to repositories or cases in which the repositories are not in Github. In the case of NVD, the main explanation is the fact that most vulnerabilities do not have any repository associated. Meanwhile in the case of OSV, we verified, as shown in table 6.1, that there is a large number of repositories in `git.kernel.org`.

The presence of vulnerabilities to which the language was not identified does not invalidate the results of this study, as it is not necessary for us to get this information for all the entries. Furthermore, as we will see, most of the vulnerabilities which contain fix commit data have also links to Github repositories and we were able to obtain the main language for most of these repositories.

We also considered the possibility of cloning the repositories to identify the languages instead of using an API. This would enable us to analyse repositories in other domains and even identify the specific languages used in the fix commits. However, that would require more development and would highly increase the processing time, since it would be necessary to clone thousands of repositories. We decided that for our goals, the precision and coverage of the data obtained using Github API is enough.

As we can see in the charts, some of the databases have one or two predominant language, while others, like OSV and NVD, are more diverse. Some databases, like RustSec and PyPA, are focused in only one language each, because they were specifically made to cover projects in the scope of these languages. Other Databases, like NVD and OSV, have a more broad scope covering vulnerabilities in many languages. The OSS-Fuzz tool supports C/C++, Rust, Go, Python and Java, but the database contains mostly C and C++ projects.

The data presented in this section can help choosing the databases to be used in the context

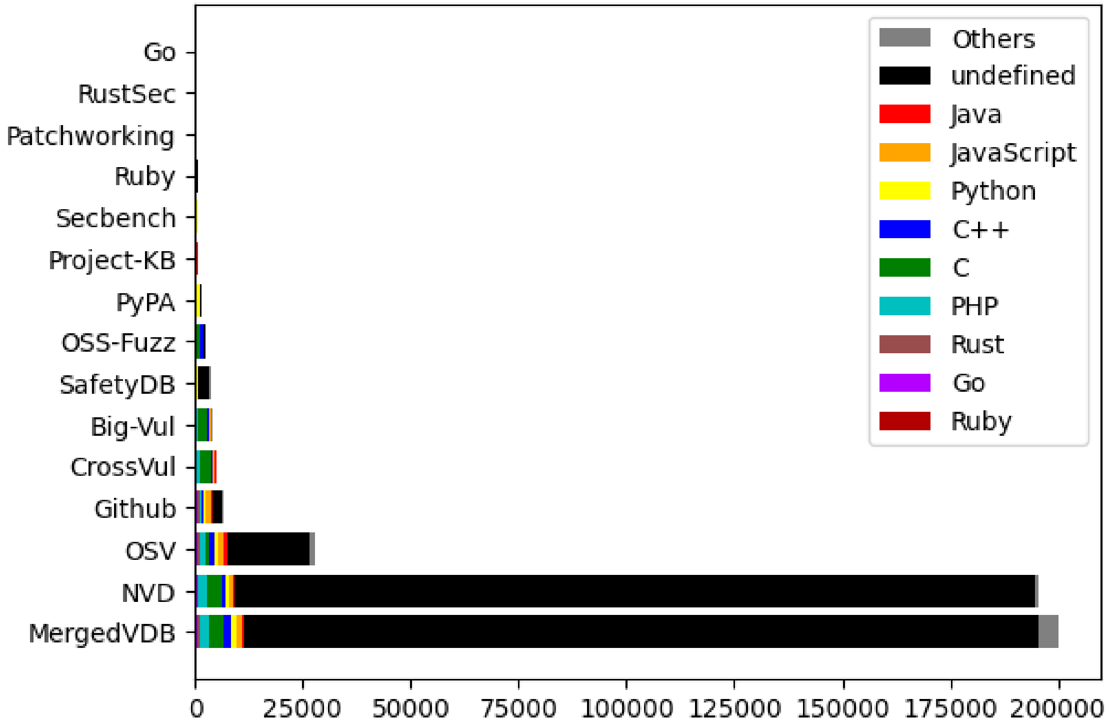


Figure 6.1: Vulnerabilities’ Languages per Database

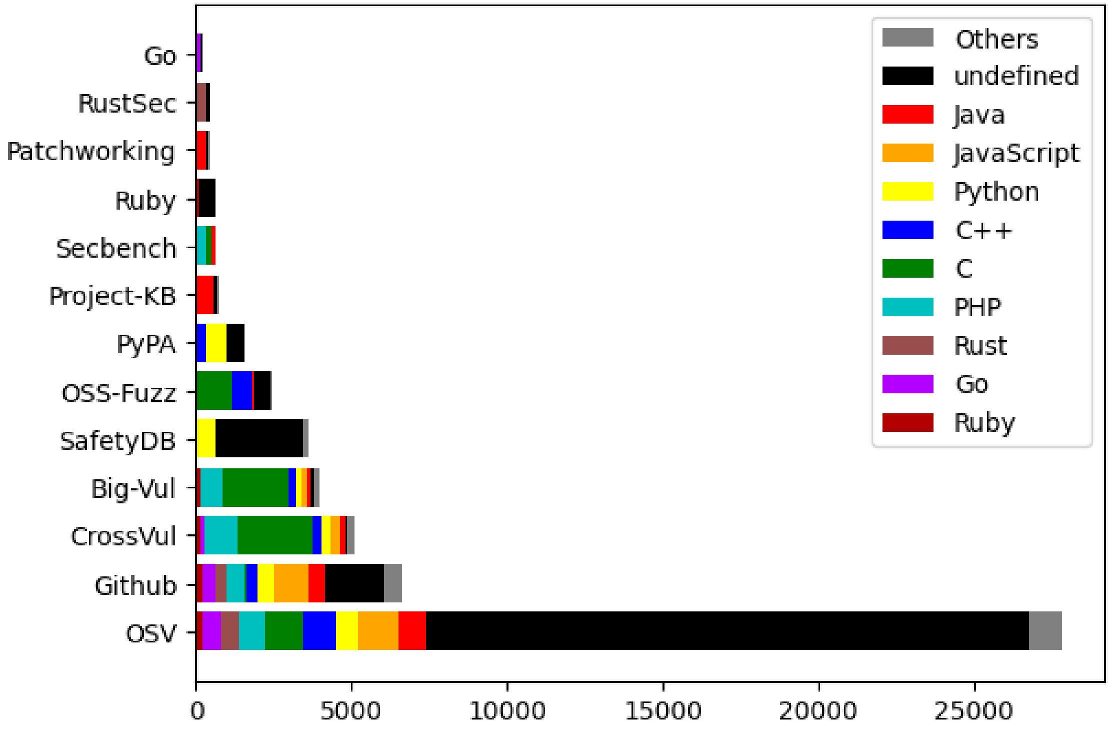


Figure 6.2: Vulnerabilities’ Languages per Database (except NVD)

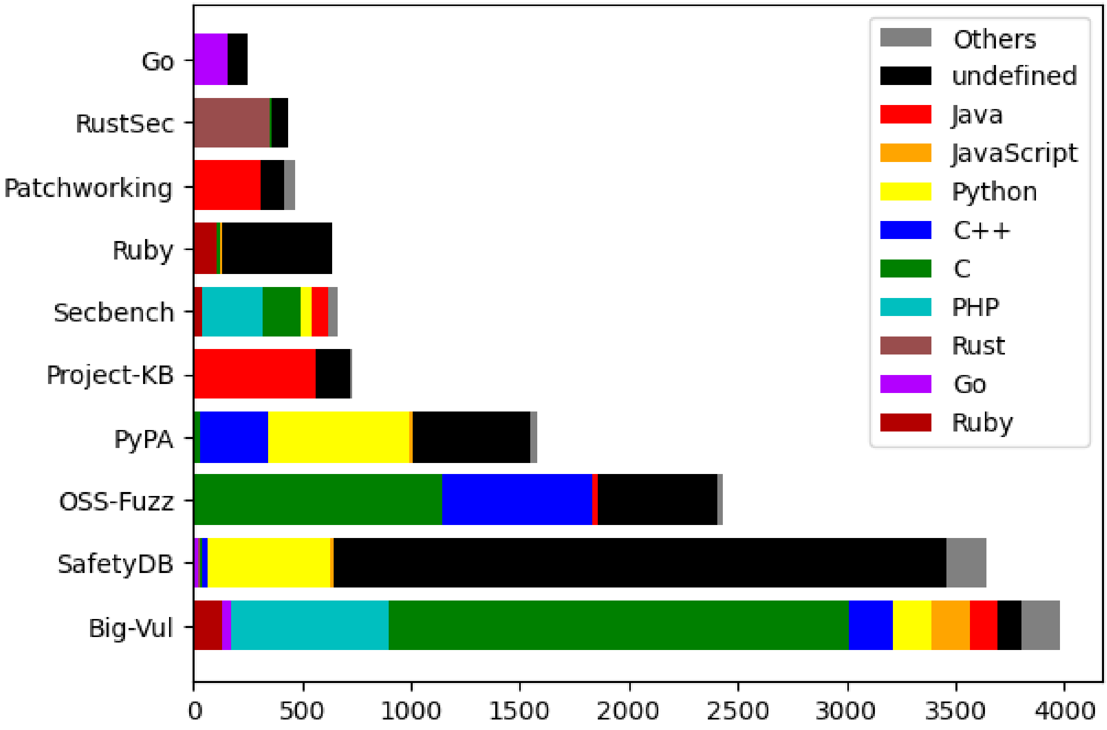


Figure 6.3: Vulnerabilities' Languages per Database (only the smallest databases)

of a specific project. For example, for Java projects, Patchworking, Project-KB, OSV and NVD cover almost all the vulnerabilities. Github Advisories also contains Java vulnerabilities, but they are also imported to OSV.

6.4 Overlap

We measured the overlaps between the databases by comparing vulnerability ids present in each of them. We do this by comparing the Id's in different databases. Some of the databases often have multiple identifiers for the same vulnerability. We use this information to conclude that two vulnerabilities that have one or more aliases in common are the same.

We present the overlap between the pairs of databases in the form of a matrix presented in tables 6.3 and 6.4.

Notice that the matrix is not symmetric because our method of comparison can lead to small inconsistencies: sometimes a vulnerability is represented by two different identifiers in one system and only one in another system. This leads our program to count a different number of overlapping vulnerabilities in each side. We decided to keep it this way because this variation is relatively small and we do not have solid criteria to decide which of the databases should be considered correct.

With the exception of OSS-Fuzz, many of the databases have a high number of entries in common with others. This is true principally in the cases in which a database uses the other as data source or has a data source in common. The few cases in which the overlap is close to 0

	NVD	CrossVul	Big-Vul	OSV	RustSec	PyPA	Go
NVD	195571	5131	3977	22716	317	1572	236
CrossVul	5131	5131	3699	2108	5	205	32
Big-Vul	3977	3699	3977	1539	0	103	18
OSV	13423	1358	850	27783	464	1481	249
RustSec	275	5	0	434	434	4	0
PyPA	1572	205	103	1578	6	1579	1
Go	220	30	17	251	0	1	251
OSS-Fuzz	0	0	0	2430	0	0	0
Github	5492	658	270	6505	291	927	98
Ruby	558	117	84	494	0	2	0
Project-KB	725	79	57	545	0	20	2
Patchworking	471	87	83	303	0	0	0
Secbench	170	60	63	94	0	4	0
SafetyDB	1796	239	116	1566	12	1088	7

Table 6.3: Overlaps Between Pairs of Databases

happen when we compare databases that focus on different languages, like Patchworking (focused on Java) and RustSec (focused on Rust).

The existence of overlapping entries is interesting for because we will use them to calculate a few measurements in section 6.5 and merge the entries in the intersections of the databases to get more complete data.

6.5 Consistency Between Databases

A concern regarding the data in all databases is whether it is possible to rely on them for identifying vulnerable code. One problem that might be present is that the commits referenced by the database might not be fixing the vulnerability, leading to false positives, that is, code is incorrectly identified as vulnerable. Another possible problem is that the entry may not reference all the commits that were necessary to fix the vulnerability, which leads to false negatives.

It is impossible to know with certainty what databases are more accurate without manually verifying all the commits. However, as a way to help understanding the reliability and completeness of the data in each database, we developed specific measurements that compare data of pairs of databases.

The first measure is what we call consistency. When two databases have entries in common and the entries in both databases have commit information, it is possible to compare them to check if they have consistent information. To measure the consistency between a pair of databases, we count the number of entries for which the commit data is exactly the same and divide it by the total number of entries that contain commit data in both databases. The results for commit consistency are presented in figure 6.4.

	OSS-Fuzz	Github	Ruby	Project-KB	Patchworking	Secbench	SafetyDB
NVD	0	5492	557	725	471	170	1796
CrossVul	0	658	116	79	87	60	239
Big-Vul	0	270	83	57	83	63	116
OSV	2430	6307	439	485	246	79	1370
RustSec	0	252	0	0	0	0	10
PyPA	0	920	2	20	0	4	1089
Go	0	90	0	1	0	0	7
OSS-Fuzz	2430	0	0	0	0	0	0
Github	0	6622	371	414	104	28	1005
Ruby	0	370	637	1	0	27	16
Project-KB	0	414	1	729	206	0	61
Patchworking	0	104	0	206	471	2	4
Secbench	0	28	27	0	2	662	4
SafetyDB	0	1005	16	61	4	4	3643

Table 6.4: Overlaps Between Pairs of Databases

In the following heatmaps, in figures 6.4, 6.5 and 6.6, we display the results for our measurements in the following way: in each square the first number is the numerator (number of consistent entries), the second is the denominator (number of overlapping entries) and the third is the value represented as percentage.

A question that arises from these results is whether the inconsistencies come from entries that point to different commits in the same repositories or commits in different repositories. To answer this question we also measured in the overlap between the databases how many entries pointed to the same repositories or different repositories. The results of this measurements are displayed in figure 6.5.

The last measure is what we call completeness. Similarly to the previous one, in this measure we use the intersection between a pair of databases, but in this case we count the number of entries in which the set of commits in one of them is contained in the other. The entry that contains all the commits referenced in the other database is considered complete. A higher value in completeness may indicate that for most entries a database contains all the relevant commits. A lower value may indicate that the database lacks relevant commit data, but may also indicate that it is being compared to a database that contains irrelevant commits. We see the results in figure 6.6

To help comparing the reliability of the databases we display in table 6.5 whether the databases are manually curated by humans, and whether they have structured commit data. We use `Y`, `N` and `P` representing *Yes*, *No* and *Partially* The manually curated ones are presumably more reliable. The ones that do not have structured data about commits may present problems because we relied on extraction of URL's from fields that are not necessarily always dedicated to fix commits. However it is hard to draw conclusions from it since even the databases that contain manually curated and

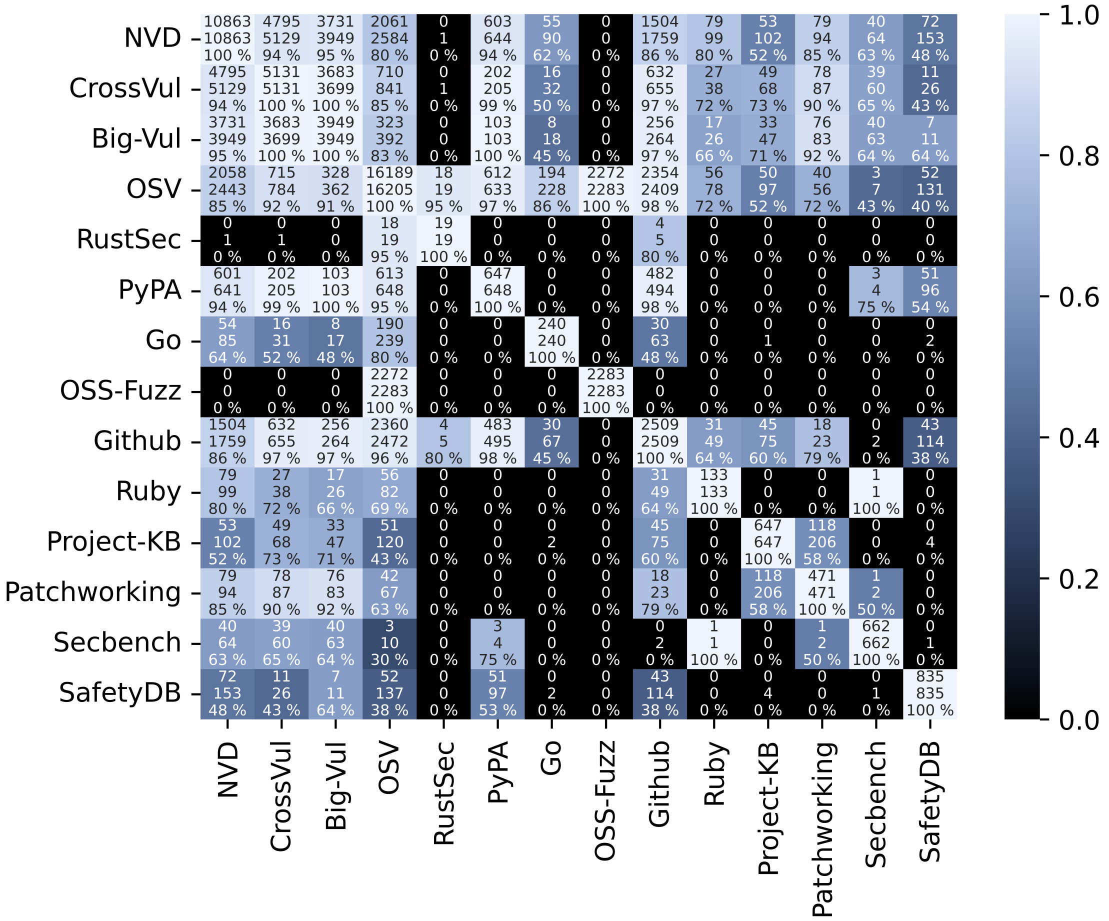


Figure 6.4: Commit Consistency Heatmap

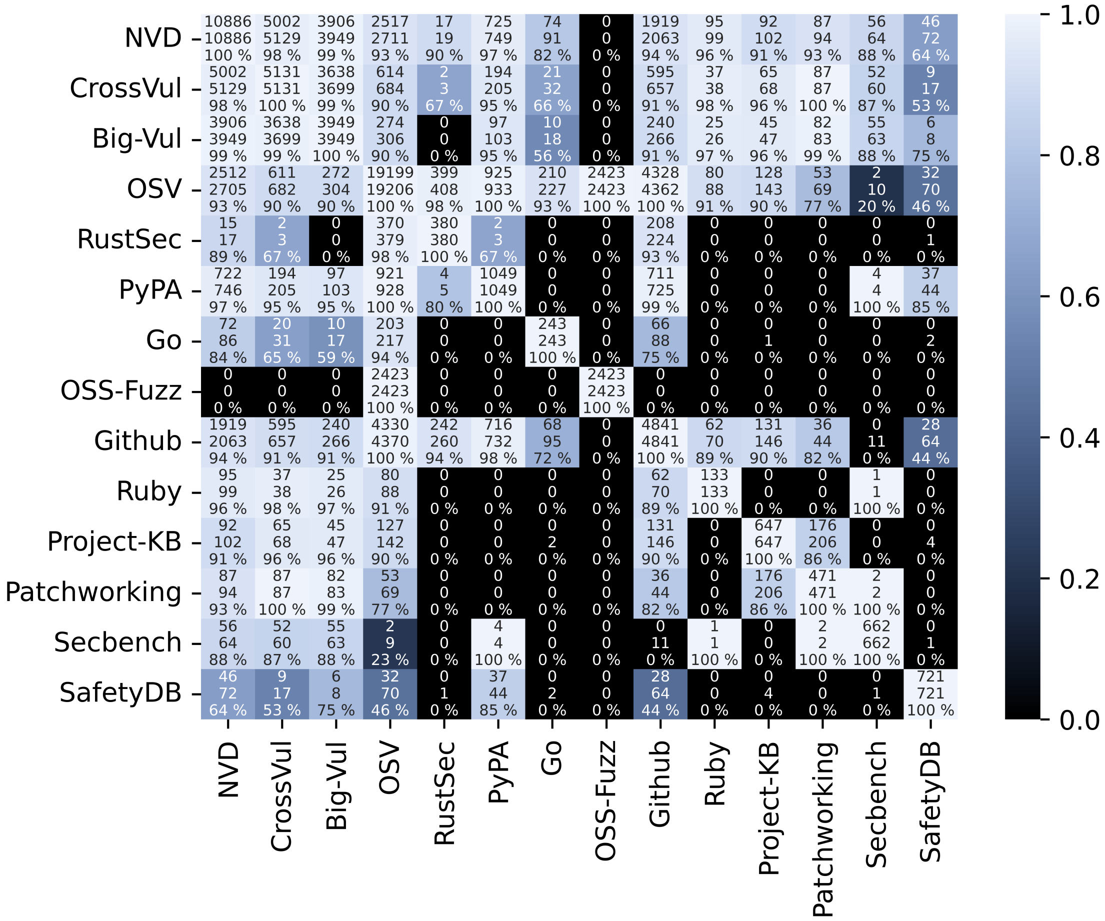
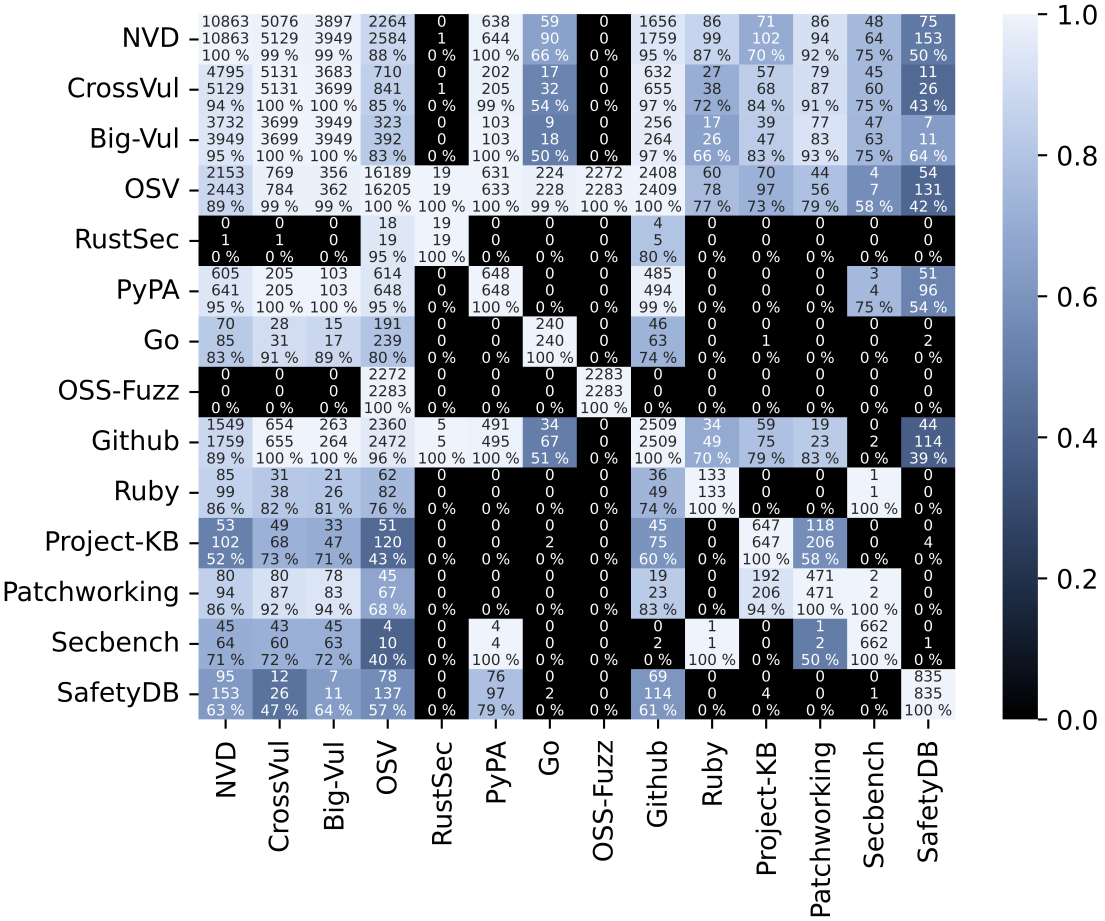


Figure 6.5: Repository Consistency Heatmap



	NVD	CrossVul	Big-Vul	OSV	RustSec	PyPA	Go	OSS-Fuzz	Github	Ruby	Project-KB	Patchworking	Secbench	SafetyDB
NVD	10863 10863 100 %	5076 5129 99 %	3897 3949 99 %	2264 2584 88 %	0 1 0 %	638 644 100 %	59 90 66 %	0 0 0 %	1656 1759 95 %	86 99 87 %	71 102 70 %	86 94 92 %	48 64 75 %	75 153 50 %
CrossVul	4795 5129 94 %	5131 5131 100 %	3683 3699 100 %	710 841 85 %	0 1 0 %	202 205 99 %	17 32 54 %	0 0 0 %	632 655 97 %	27 38 72 %	57 68 84 %	79 87 91 %	45 60 75 %	11 26 43 %
Big-Vul	3732 3949 95 %	3699 3699 100 %	3949 3949 100 %	323 392 83 %	0 0 0 %	103 103 100 %	9 18 50 %	0 0 0 %	256 264 97 %	17 26 66 %	39 47 83 %	77 83 93 %	47 63 75 %	7 11 64 %
OSV	2153 2443 89 %	769 784 99 %	356 362 99 %	16189 16205 100 %	19 19 100 %	631 633 100 %	224 228 99 %	2272 2283 100 %	2408 2409 100 %	60 78 77 %	70 97 73 %	44 56 79 %	4 7 58 %	54 131 42 %
RustSec	0 1 0 %	0 1 0 %	0 0 0 %	18 19 95 %	19 19 100 %	0 0 0 %	0 0 0 %	0 0 0 %	4 5 80 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %
PyPA	605 641 95 %	205 205 100 %	103 103 100 %	614 648 95 %	0 0 0 %	648 648 100 %	0 0 0 %	0 0 0 %	485 494 99 %	0 0 0 %	0 0 0 %	0 0 0 %	3 4 75 %	51 96 54 %
Go	70 85 83 %	28 31 91 %	15 17 89 %	191 239 80 %	0 0 0 %	0 0 0 %	240 240 100 %	0 0 0 %	46 63 74 %	0 0 0 %	0 1 0 %	0 0 0 %	0 0 0 %	0 2 0 %
OSS-Fuzz	0 0 0 %	0 0 0 %	0 0 0 %	2272 2283 100 %	0 0 0 %	0 0 0 %	0 0 0 %	2283 2283 100 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %
Github	1549 1759 89 %	654 655 100 %	263 264 100 %	2360 2472 96 %	5 5 100 %	491 495 100 %	34 67 51 %	0 0 0 %	2509 2509 100 %	34 49 70 %	59 75 79 %	19 23 83 %	0 2 0 %	44 114 39 %
Ruby	85 99 86 %	31 38 82 %	21 26 81 %	62 82 76 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %	36 49 74 %	133 133 100 %	0 0 0 %	0 0 0 %	1 1 100 %	0 0 0 %
Project-KB	53 102 52 %	49 68 73 %	33 47 71 %	51 120 43 %	0 0 0 %	0 0 0 %	0 2 0 %	0 0 0 %	45 75 60 %	0 0 0 %	647 647 100 %	118 206 58 %	0 0 0 %	0 4 0 %
Patchworking	80 94 86 %	80 87 92 %	78 83 94 %	45 67 68 %	0 0 0 %	0 0 0 %	0 0 0 %	0 0 0 %	19 23 83 %	0 0 0 %	192 206 94 %	471 471 100 %	2 2 100 %	0 0 0 %
Secbench	45 64 71 %	43 60 72 %	45 63 72 %	4 10 40 %	0 0 0 %	4 4 100 %	0 0 0 %	0 0 0 %	0 2 0 %	1 1 100 %	0 0 0 %	1 2 50 %	662 662 100 %	0 1 0 %
SafetyDB	95 153 63 %	12 26 47 %	7 11 64 %	78 137 57 %	0 0 0 %	76 97 79 %	0 2 0 %	0 0 0 %	69 114 61 %	0 0 0 %	0 4 0 %	0 0 0 %	0 1 0 %	835 835 100 %

Figure 6.6: Completeness Heatmap

structured data present some significant inconsistencies between each other.

	NVD	OSV	Github	Project-Kb	CrossVul	Big-Vul	OSS-Fuzz	Go	RustSec	Ruby	Safety DB	Secbench	Patchworking	PyPA
Manually Curated	Y	P	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	P
Structured Commit Data	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y

Table 6.5: Manually Curated Structured Data

In the heatmaps we represent the measurements as fractions, with absolute values of the number of entries, as well as the percentages. Notice that in all of these heatmaps there are pairs of databases that do not have any overlap, and in these cases it was not possible to calculate the consistency or completeness. For example, OSS-Fuzz only had overlap with OSV, therefore it was not possible to calculate the measurements for this database. There are also cases in which there is overlap between the databases but it is just a very small number of entries, and the overlap can be calculated, but is not a very significant measurement. For example, there is an overlap of only 2 vulnerabilities with commits between Secbench and Patchworking. We calculated 50% of consistency, but since this value is based on just 2 entries, we can not be sure whether this is representative of the reliability of these databases.

The databases that are imported by OSV had in general high values of consistency with it. The small inconsistencies can be explained either by delays in OSV to incorporate the updates from other databases, or by the fact that OSV considers multiple sources, and consequently may have entries that inconsistent data in different sources.

Since Big-Vul, CrossVul and our work present similar methods for extracting information from NVD, it is interesting to look at the values of consistency and completeness between these 3 databases. Notice that the values displayed here for NVD are relative to the commits and repositories obtained using our method.

Unsurprisingly, since the methods are similar, the values are relatively high, with at least 97% of consistency between all the pairs. The small inconsistencies between them are explained by implementation details, such as the consideration of pull request and issues links and repositories in other domains besides Github. We found that our method yields higher values in completeness and lower values in consistency. By manually looking at samples of inconsistent entries, we found that this happens mostly due to pull request links that are included in our database and not in the others. We manually verified some of these pull requests and confirmed that many of them have relevant security fixes, so we decided to keep them. This difference in our method can lead to more false negatives, as the pull requests may have irrelevant commits, but harnesses a higher part of the information available in NVD, leading to less false negatives.

Safety DB is a database with particularly low values in consistency. Examining the entries in this database we verified that many of them have links to commits or pull requests that do not directly fix a vulnerability, but change a dependency in project that imports the one that contains the vulnerability. That also causes Safety DB to have the lowest values in repository consistency.

When comparing Project-KB to the 3 databases obtained from NVD (CrossVul, Big-Vul and our own approach) we see that Project-KB presents lower values in completeness. For example when compared to the entries we obtained from NVD, Project-KB has completeness of 52% and NVD has 70%. We found a few cases in which this is explained by the fact that we extracted all commits from pull request links in the NVD and Project-KB had only the commits that were relevant to the vulnerability. That may indicate a higher number of false negatives in the automated approaches, but we can not guarantee that this is always the case.

Curiously, we found that Project-KB and OSV have only 43% of consistency with each other in spite of both having fields for fix commits and being at least partially curated by humans. The vulnerabilities in the overlap between OSV and Project-KB come from Github Advisory Database and GSD. In this comparison, OSV presents 73% of completeness and Project-KB only 43%, meaning that there are many cases in which OSV lists more commits than Project-KB. We do not know which one is more accurate. OSV has approximately the same consistency with the 3 automated approaches as Project-KB, but with higher values of completeness. The higher completeness in OSV may be partially explained by the fact that some entries in OSV have more than one CVE, therefore representing more than one entry in NVD or Project-KB.

Overall it is hard to take final conclusions since there are many cases in which databases present relatively low values of consistency in spite of being manually curated. We can not tell which database is more reliable. It is important to notice that the manual verification process will yield different results when performed by different organisations with different methodologies. However, we can at least notice that, with the exception of Safety DB, most databases have values higher than 50% in almost all the significant comparisons.

6.6 New Statements for Project-KB

As described in section 4.1 Project-KB requires each of its entries to contain information about fix commits or artifacts affected by the vulnerability. The information presented in table 6.2 allows us to conclude that the data we extracted would be enough to generate 27163 statements. However, we decided to focus only on the ones with commit data, reducing this number to 24787.

In figure 6.7 we see the distribution of languages of the repositories affected by the vulnerabilities corresponding to the statements coming from each repository. *MergedVDB* represents the database obtained by joining all the others.

Since the resulting database aggregates data from all the others, the values of completeness is always 100%. Since it includes all the commits in the other databases, it contains more false positives and less false negatives than the all others.

We made a Venn diagram, displayed in figure 6.8 to compare CrossVul, Big-Vul and the entries we obtained from NVD. We can see that we were able to extract more vulnerabilities with commits from NVD than CrossVul and Big-Vul. This is mostly explained by the fact that our study is more

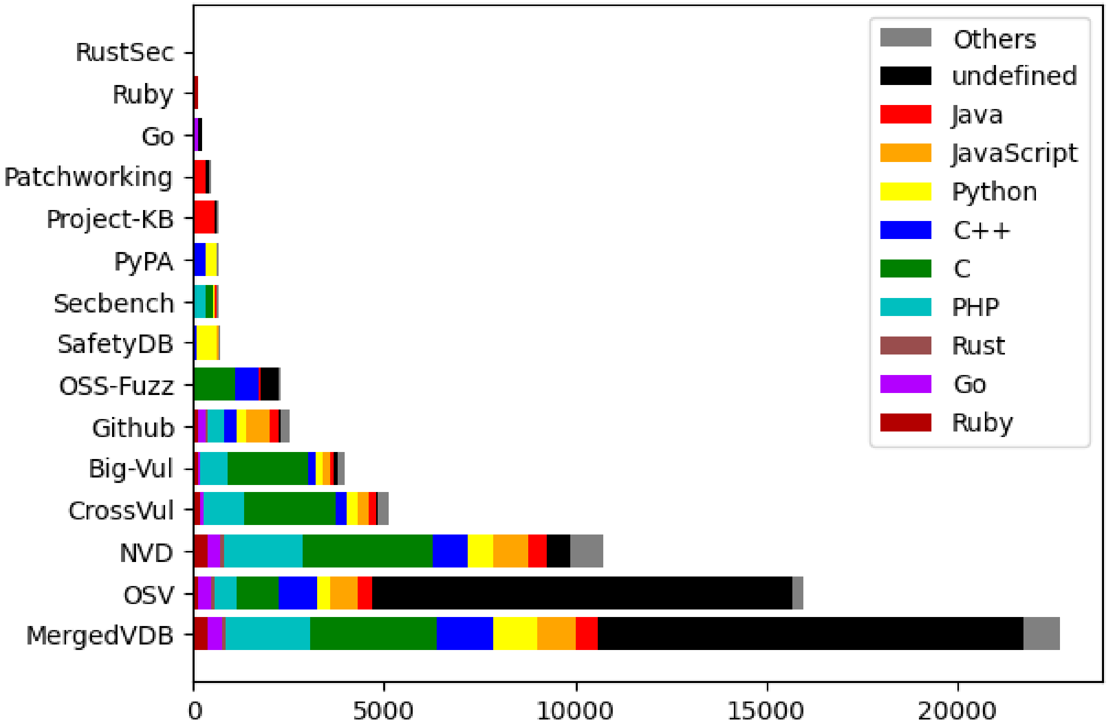


Figure 6.7: Languages of Statements Coming from Each VDB

recent and now the NVD contains more entries with commit data. Other explanation is that each of the 3 approaches implements slightly different methods for extracting data from NVD. For example, they may consider only commit links, or also pull requests, and look for URL's in other domains besides Github.

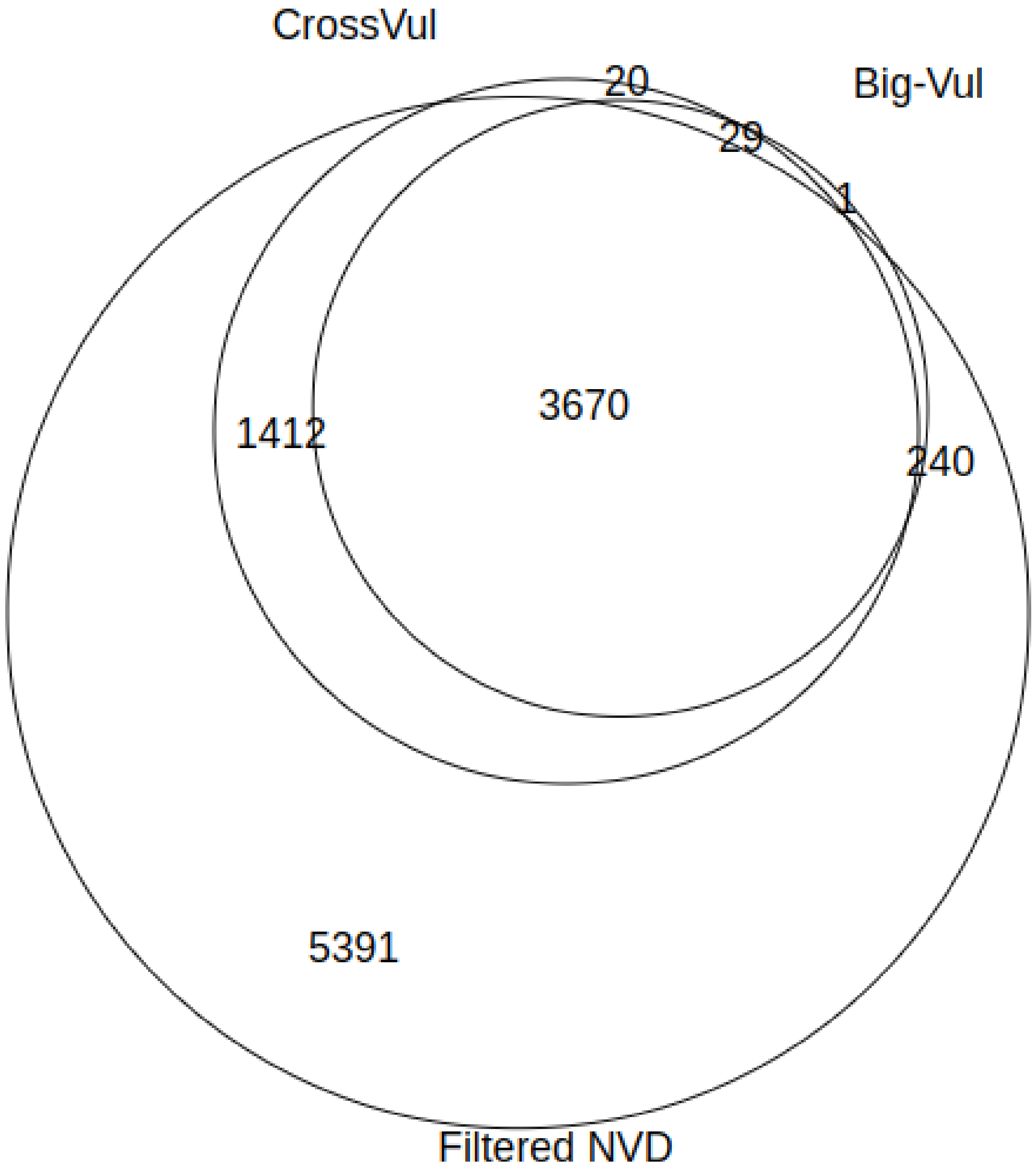


Figure 6.8: Venn Diagram

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this study we presented an overview of the current state of vulnerability databases for open source software focusing on the presence of code-level data. Our analysis allowed us to have insights on the scope, size and reliability of each of them, and provides objective information that can be used by software developers and security researchers to determine which databases are more suited for each use case.

We also developed a pipeline to aggregate data from many of these source in a single database with fix commits for each vulnerability. The final result is, as far as we know, the largest vulnerability database with fix commit data, and its pipeline allows us to continue to expand it as its data sources are updated. We generated a total of N entries that can potentially be used by Project-Kb and Eclipse Steady.

Furthermore, we developed improvements in Eclipse Steady which greatly reduce its processing time, making it easier to import a larger number of vulnerabilities, in case the new entries are used.

7.2 Future Work

Other heuristics can be developed in the program to decrease the number of false positives and increase the number of identified commits. For example, an alternative strategy for detecting fix commits would be to look for them directly in the git repositories instead of using only the information available in the databases. Then the fix commits could be identified with the help of other heuristics, such as looking for the vulnerability identifier in the commit messages. Heuristics like this can be found in works such as Patch-Finder [18], and in Prospector [42], a tool that uses the description of a vulnerability in natural language to identify and rank commits that potentially fix it.

Another possible improvement would be to to scrap web pages with vulnerability fix information. We found in the databases many links that contain relevant commit data but were not used by our program. [1]

Due to time limitations, some databases were not included in the pipeline that could be considered in the future. Some examples include GSD, Friends Of PHP Security Advisories [13] and Mozilla Foundation Security Advisories [22].

[1] One example of link that potentially contains information about a vulnerability fix is `https://go-review.googlesource.com/c/go/+/417062/`

References

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021.

[2] Cloud Security Alliance. GlobalSecurityDatabase. Available at https://globalsecuritydatabase.org/, Accessed last time in September 2022. Last accessed 18 September 2022.

[3] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. Commit2vec: Learning distributed representations of code changes. In *SN Computer Science*, volume 2, 2021.

[4] Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Matias Martinez, and Corrado A. Visaggio. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology*, 142:106745, 2022.

[5] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. *Code-Based Vulnerability Detection in Node.Js Applications: How Far Are We?*, page 1199–1203. Association for Computing Machinery, New York, NY, USA, 2020.

[6] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners - a study amp; test suite. *IEEE Transactions on Software Engineering*, pages 1–1, 2021.

[7] Debricked. About Security | Documentation | Debricked. Available at https://debricked.com/docs/security/security-about.html#data-sources, Accessed last time in September 2022. Last accessed 18 September 2022.

[8] Debricked. Vulnerability Database. Available at https://debricked.com/vulnerability-database/, Accessed last time in September 2022. Last accessed 18 September 2022.

[9] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[10] FASTEN. Main -XWiki. Available at https://www.fasten-project.eu/view/Main, Accessed last time in September 2022. Last accessed 18 September 2022.

[11] fasten project. Vulnerability Plugin. Available at https://github.com/fasten-project/vulnerability-producer/blob/develop/docs/documentation.md, Accessed last time in September 2022. Last accessed 18 September 2022.

[12] FIRST. CVSS v3.1 Specification Document. Available at https://www.first.org/cvss/v3.1/specification-document, Accessed last time in March 2022. Last accessed 6 March 2022.

[13] FriendsOfPHP. PHP Security Advisories Database. Available at https://github.com/FriendsOfPHP/security-advisories, Accessed last time in September 2022. Last accessed 18 September 2022.

[14] GitHub. Browsing security advisories in the GitHub Advisory Database. Available at https://docs.github.com/en/code-security/dependabot/dependabot-alerts/browsing-security-advisories-in-the-github-advisory-database#about-the-github-advisory-database, Accessed last time in June 2022. Last accessed 18 June 2022.

[15] GitHub. GitHub Advisory Database. Available at https://github.com/advisories, Accessed last time in September 2022. Last accessed 18 September 2022.

[16] Google. Go Vulnerability Management - The Go Programming Language. Available at https://go.dev/security/vulndb/, Accessed last time in June 2022. Last accessed 22 June 2022.

[17] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. Available at https://github.com/google/oss-fuzz, Accessed last time in September 2022. Last accessed 18 September 2022.

[18] Daan Hommersom, Antonino Sabetta, Bonaventura Coppola, and Damian A. Tamburri. Automated mapping of vulnerability advisories onto their fix commits in open source repositories, March 2021.

[19] Arvinder Kaur and Ruchikaa Nayyar. A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[20] Eduard Kovaks. OSVDB Shut Down Permanently. Available at https://www.securityweek.com/osvdb-shut-down-permanently, Accessed last time in September 2022. Last accessed 18 September 2022.

[21] Kyriakos Kritikos, Kostas Magoutis, Manos Papoutsakis, and Sotiris Ioannidis. A survey on vulnerability assessment tools and databases for cloud-based web applications. *Array*, 3-4:100011, 2019.

[22] Mozilla. Mozilla Foundation Security Advisories. Available at https://www.mozilla.org/en-US/security/advisories/, Accessed last time in September 2022. Last accessed 18 September 2022.

[23] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 29–41, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. *CrossVul: A Cross-Language Vulnerability Dataset with Commit Data*, page 1565–1569. Association for Computing Machinery, New York, NY, USA, 2021.

[25] NIST. NVD - Home. Available at https://nvd.nist.gov/, Accessed last time in February 2022. Last accessed 27 February 2022.

[26] NIST. Security Content Automation Protocol | CSRC. Available at https://csrc.nist.gov/projects/security-content-automation-protocol/, Accessed last time in March 2022. Last accessed 4 March 2022.

[27] npm. npm Docs. Available at https://docs.npmjs.com/cli/v8/commands/npm-audit, Accessed last time in September 2022. Last accessed 18 September 2022.

[28] Open Web Application Security Project. OWASP Top 10. Available at https://owasp.org/www-project-top-ten/, Accessed last time in January 2022, 2021.

[29] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420, 2015.

[30] Serena Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cedric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019.

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

[32] Open Web Application Security Project. OWASP Dependency-Check Project. Available at https://owasp.org/www-project-dependency-check/, Accessed last time in February 2022. Last accessed 27 February 2022.

[33] PyPA. Advisory database for packages published on https://pypi.org. Available at https://github.com/pypa/advisory-database, Accessed last time in June 2022. Last accessed 22 June 2022.

[34] PyPA. PyPA Goals - PyPA documentation. Available at https://www.pypa.io/en/latest/future, Accessed last time in June 2022. Last accessed 22 June 2022.

[35] PyPA. Python Packaging Authority - PyPA documentation. Available at https://www.pypa.io/en/latest/, Accessed last time in June 2022. Last accessed 22 June 2022.

[36] PyUP. Python Dependency Security. Available at https://pyup.io/, Accessed last time in September 2022. Last accessed 18 September 2022.

[37] pyupio. Safety DB. Available at https://github.com/pyupio/safety-db, Accessed last time in September 2022. Last accessed 18 September 2022.

[38] Sofia Reis and Rui Abreu. SECBENCH: A database of real security vulnerabilities. In Martin Gilje Jaatun and Daniela Soares Cruzes, editors, *Proceedings of the International Workshop on Secure Software Engineering in DevOps and Agile Development co-located with the 22nd European Symposium on Research in Computer Security (ESORICS 2017), Oslo, Norway, September 14, 2017*, volume 1977 of *CEUR Workshop Proceedings*, pages 69–85. CEUR-WS.org, 2017.

[39] rubysec. Ruby Advisory Database. Available at https://github.com/rubysec/ruby-advisory-db, Accessed last time in September 2022. Last accessed 18 September 2022.

[40] Rust. Security Policy - Rust Programing Language. Available at https://www.rust-lang.org/policies/security, Accessed last time in September 2022. Last accessed 18 September 2022.

[41] RustSec. RustSec Advisory Database. Available at https://rustsec.org/, Accessed last time in September 2022. Last accessed 18 September 2022.

[42] SAP. Prospector. Available at https://github.com/SAP/project-kb/tree/master/prospector, Accessed last time in September 2022. Last accessed 18 September 2022.

[43] Henrik Plate Serena Elisa Ponta and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 2020.

[44] Snyk. Open source security management. Available at https://snyk.io/product/open-source-security-management. Last accessed 27 February 2022.

[45] Snyk. Snyk Intelligence Security. Available at https://security.snyk.io/, Accessed last time in March 2022. Last accessed 4 March 2022.

[46] Snyk. Snyk Vulnerability Database. Available at https://security.snyk.io/, Accessed last time in March 2022. Last accessed 4 March 2022.

[47] Snyk. State of open source security report 2020. Available at https://go.snyk.io/rs/677-THP-415/images/State%20OfOpenSourceSecurityReport2020.pdf, Accessed last time in January 2022, 2020.

[48] Anshu Tripathi and Umesh Kumar Singh. Taxonomic analysis of classification schemes in vulnerability databases. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 686–691, 2011.

[49] WhiteSource. Dynamic Application Security Testing: DAST Basics. Available at hhttps://www.whitesourcesoftware.com/resources/blog/dast-dynamic-application-security-testing/, Accessed last time in March 2022. Last accessed 6 March 2022.

[50] WhiteSource. Open Source Vulnerability Database. Available at https://vuln.whitesourcesoftware.com/vulnerability-database/, Accessed last time in June 2022. Last accessed 9 June 2022.

[51] Awad A. Younis, Yashwant K. Malaiya, and Indrajit Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014*

IEEE 15th International Symposium on High-Assurance Systems Engineering, pages 1–8, 2014.

[52] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu 0003. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Available at `https://sites.google.com/view/devign`, Accessed last time in September 2022.

[53] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu 0003. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Edward A. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 10197–10207, 2019.