Bachelor thesis

A large multi-language dataset of open-source software vulnerabilities and their fixes

by **Kevin Washington da Silva Lira**

advised by
Prof. Ph.D. Baldoino Fonseca dos Santos Neto

Federal University of Alagoas
Computing Institute
Maceió, Alagoas
December 04, 2023

FEDERAL UNIVERSITY OF ALAGOAS
Computing Institute

A LARGE MULTI-LANGUAGE DATASET OF OPEN-SOURCE SOFTWARE VULNERABILITIES AND THEIR FIXES

Bachelor Thesis submitted to the Computing Institute from Federal University of Alagoas as a partial requirement to obtain the bachelor degree in Computer Engineering.

Kevin Washington da Silva Lira

Advisor: Prof. Ph.D. Baldoino Fonseca dos Santos Neto

Examining Board:

Ícaro Bezerra Queiroz de Araújo	Prof. Ph.D., UFAL
Davy de Medeiros Baia	Prof. Ph.D., UFAL

Maceió, Alagoas
December 04, 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

L768l Lira, Kevin Washington da Silva.
A large multi-language dataset of open-source software vulnerabilities and their fixes / Kevin Washington da Silva Lira. – 2023.
38 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.
Monografia (Trabalho de conclusão de curso em Engenharia de Computação) - Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2023.

Bibliografia: f. 36-38.

1. Segurança da informação. 2. Ameaça. 3. Vulnerabilidades (Informática). 4. Software de código aberto. I. Título.

CDU: 004.056.5

,

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL

Instituto de Computação – IC

Campus A. C. Simões - Av. Lourival de Melo Mota, BL 12
Tabuleiro do Martins, Maceió/AL - CEP: 57.072-970
Telefone: (082) 3214-1401

Trabalho de Conclusão de Curso - TCC

Formulário de Avaliação

Nome do Aluno

K	E	V	I	N		W	A	S	H	I	N	G	T	O	N		D	A
S	I	L	V	A		L	I	R	A									

Nº de Matrícula

1	7	1	1	1	4	1						-	7

Título do TCC (Tema)

A large multi-language dataset of open-source software vulnerabilities and their fixes

Banca Examinadora

Baldoino Fonseca dos Santos Neto
Nome do Orientador

Documento assinado digitalmente
gov.br BALDOINO FONSECA DOS SANTOS NETO
Data: 05/12/2023 14:10:18-0300
Verifique em https://validar.iti.gov.br

Assinatura

Ícaro Bezerra Queiroz de Araújo
Nome do Professor

Documento assinado digitalmente
gov.br ICARO BEZERRA QUEIROZ DE ARAUJO
Data: 05/12/2023 09:46:43-0300
Verifique em https://validar.iti.gov.br

Davy de Medeiros Baia
Nome do Professor

Documento assinado digitalmente
gov.br DAVY DE MEDEIROS BAIA
Data: 05/12/2023 11:47:27-0300
Verifique em https://validar.iti.gov.br

Data da Defesa

04/ Dez/ 2023

Nota Obtida

10,00 (dez)

Observações

Coordenador do Curso
De Acordo

Assinatura

Acknowledgments

I would like to express my deepest gratitude to Prof. Baldoino Fonseca for his invaluable guidance and support during the preparation of this work. His wisdom, advice, and availability have been fundamental to my academic and personal growth. I would also like to sincerely thank Prof. Ícaro Araújo and Prof. Davy Baia for their participation in the chair and their valuable contributions to the improvement of this work.

To my parents, I am immensely grateful for the love, support and sacrifices they have made to help me reach this important milestone. Their constant encouragement has been the foundation for everything I have accomplished in my life.

To my classmates, especially the members of Pareias: Álvaro, Augusto, Hugo, João Arthur, Sofia, Rafael and Valério, walking this path with you has been an incredibly enriching experience. Thank you for the discussions, the mutual support and all the fun moments over the years.

To my beloved partner Karoliny, your care and support were essential during this challenging journey. Thank you for being by my side, sharing the challenges and celebrating the successes.

Kevin Washington da Silva Lira

Apenas que... busquem conhecimento.
ET Bilu

Resumo

No cenário atual do aumento progressivo da adoção de ferramentas digitais pela sociedade, softwares de todos os tipos e tamanhos enfrentam constantemente ameaças à sua segurança. No contexto da segurança digital, uma vulnerabilidade é definida como uma fraqueza encontrada em componentes de software e hardware que, quando explorada, resulta em um impacto negativo na confidencialidade, integridade ou disponibilidade do serviço. O processo de mitigação das vulnerabilidades de segurança presentes em softwares normalmente envolve alterações de código. Dessa forma, é necessário identificar o trecho de código que introduz uma vulnerabilidade para que seja possível realizar a implementação da correção. Este trabalho apresenta uma metodologia de identificação e extração de códigos vulneráveis em projetos de softwares *open-source* e seus respectivos patches. Para isso, é apresentada uma ferramenta que identifica as vulnerabilidades de software publicadas e extrai, de forma automática, o código associado. O dataset construído reúne vulnerabilidades de software e seus patches presentes em 3,587 projetos desenvolvidos em 58 linguagens de programação. Além disso, foram realizadas análises com o intuito de verificar a incidência das vulnerabilidades e as características dos fixes desenvolvidos nas principais linguagens do mercado.

Palavras-chave*: *Segurança da informação*; *Ameaças*; *Vulnerabilidades*; *Software de código aberto.

Abstract

In the current scenario of progressive increase in the adoption of digital tools by society, software of all types and sizes constantly faces threats to its security. In the context of digital security, a vulnerability is defined as a weakness found in software and hardware components that, when exploited, negatively impact the service's confidentiality, integrity, or availability. The process of mitigating security vulnerabilities present in software typically involves code changes. Therefore, it is necessary to identify the snippet of code that introduces a vulnerability to implement the correction. This work presents a methodology for identifying and extracting vulnerable codes in *open-source* software projects and their patches. For this purpose, a tool is presented that identifies published software vulnerabilities and automatically extracts the associated code. The constructed dataset combines software vulnerabilities and their fixes in 3,587 projects developed in 58 programming languages. Furthermore, analyses were carried out to verify the incidence of vulnerabilities and the characteristics of fixes developed in the main languages on the market.

Keywords*: *Information security*; *Threats*; *Vulnerabilities*; *Open-source software.

List of Figures

List of Tables

List of Acronyms

CPE	Common Platform Enumeration
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DMM	Delta Maintainability Model
LLM	Large Language Model
MFA	multi-factor authentication
MITM	Man-in-the-middle
NLOC	Number of lines of code
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
SCAP	Security Content Automation Protocol
SIG-MM	SIG Maintainability Model
XSS	Cross-site scripting

Contents

Chapter 1

Introduction

In today's constantly changing digital landscape, software systems of all sizes face ongoing security threats. A software vulnerability is a security flaw, glitch, or weakness found in software code that could be exploited by an attacker [Dempsey et al., 2020] to gain unauthorized access or cause harm to a system. These vulnerabilities can occur due to a range of issues, including but not limited to coding errors, inadequate security controls, software complexity, or the use of outdated or compromised third-party components [Luo et al., 2020].

To ensure software reliability and data security, it is critical to detect and counter these vulnerabilities consistently. For example, a cross-site scripting (XSS) or SQL injection vulnerability in a Web application requires an urgent remediation strategy [Erbel and Kopniak, 2018]. These two software vulnerabilities are widely known and easily exploited by malicious users seeking unauthorized access and compromise of sensitive data. However, software vulnerabilities that require administrator-level access or are restricted to local networks are typically a lower priority because their exploitation potential is limited to a smaller subset of individuals.

These vulnerabilities are generally registered in a repository. One of these repositories is The National Vulnerability Database (NVD). NVD is a repository of vulnerability management data maintained by the United States Government's National Institute of Standards and Technology (NIST). It uses the Security Content Automation Protocol (SCAP) to represent data in a standards-based format. The NVD's primary goal is to automate vulnerability management, security measurement, and compliance. To achieve this objective, the NVD maintains databases of security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics.

Given the scenario of various programming languages and their particularities, the study of vulnerabilities in different programming languages is critical in digital security. Each programming language has its own resources and different application areas, which results in different types of vulnerabilities that can be exploited. Languages aimed at web development, such as PHP and JavaScript, are more susceptible to cross-site scripting

(XSS) or code injection attacks, while languages such as C and C++, frequently used in operating systems and embedded software, are more vulnerable to memory management failures. Understanding these differences is crucial to developing more effective and personalized solutions for each context and language. Furthermore, the in-depth study of these vulnerabilities helps in the creation of more advanced detection and correction tools, contributing to the development of secure applications.

Several previous works have proposed the creation of datasets that search for vulnerabilities associated with code snippets. In [Bhandari et al., 2021], the authors proposed a dataset of vulnerabilities collected from NVD without performing any temporal analysis or categorization by language or group of vulnerabilities. In [Fan et al., 2020], the presented dataset contains only vulnerabilities found in applications in C/C++ languages. In [Challande et al., 2022], the authors extracted and analyzed vulnerabilities from one project: the Android Open Source Project (AOSP).

In the software vulnerability life cycle context, previous work has been done to analyze the evolution of different vulnerabilities to your discovery, patching, and exploitation [Joh and Malaiya, 2017, Le et al., 2021, Lin et al., 2023, Shahzad et al., 2012]. In [Shahzad et al., 2012], there was a more extensive and in-depth analysis of the entire lifecycle of a software vulnerability, from the moment of its discovery to its publication, patching, and exploitation. However, this study only covers vulnerabilities published up to 2011, and the data sources used for the proposed analyses are no longer publicly available, which prevents an analysis of this work with more up-to-date data.

The objectives of this work are to propose a new large dataset that presents software vulnerabilities and their respective fix commits found in open-source projects in different programming languages and to perform initial analyses on the collected data to understand how these patches differ between programming languages and the different categories of vulnerabilities assessed. For that, a tool was developed responsible for collecting information about vulnerabilities published in the NVD and in various open source code repositories. The extracted data was pre-processed and filtered, and the result served as the basis for building the proposed dataset.

Aiming to validate the constructed dataset, the following research questions were defined to be answered using the data obtained: **RQ1 - How does the incidence of reported software vulnerability compare for different programming languages?** And **RQ2 - What are the characteristics of vulnerability fixes?** Both research questions are crucial to understanding the nature of vulnerabilities and their fixes in the main programming languages on the market.

In total, 11,558 vulnerabilities were collected in 3,587 projects written in 58 different programming languages with their respective vulnerable code snippets and 11,332 commits containing the fix for the vulnerabilities. By categorizing these vulnerabilities into groups and examining their evolution, this work offers the following insights: PHP

language exhibited a high prevalence of vulnerabilities typically associated with web applications and networked environments. In contrast, languages such as C and C++ present a high percentage of vulnerabilities related to memory and resource management. Vulnerabilities in the C language have fix commits of varying sizes, while languages such as Ruby or TypeScript have smaller and simpler fixes. Over the years, it was also possible to identify an increase in the average severity of reported and fixed vulnerabilities.

To contribute to the scientific community and provide data that can be used for future research in the area of information security - which includes new analyzes of the data and use as a basis for creating Large Language Models (LLMs) for detecting and predicting software vulnerabilities in multiple languages, the dataset created and the code developed for the data collection, processing and analysis tools were made publicly available.

1.1 Objectives

The main objective of this work is to present a large dataset that contains code snippets that present software vulnerabilities and their corresponding fix commits across multiple programming languages. To achieve this, the following specific objectives were proposed:

- Analyze related work to identify existing data sources on reported vulnerabilities;
- Develop a tool that collects and catalogs published software vulnerabilities;
- Develop a tool that extracts code and data associated with related projects;
- Analyze the characteristics of the data obtained.

1.2 Structure

This work has been divided into 5 chapters to describe the steps followed during the theoretical foundation, tool development, dataset assembly, and analysis. Chapter 2 presents some concepts in the computer security field that support this work and the current state of the literature. Chapter 3 describes the tool design and development process, including the technologies used during development. Chapter 4 presents the dataset obtained and some analysis of the characteristics presented by the fixes made in the main programming languages on the market. Chapter 5 contains the final conclusions of this work[1].

[1] All the algorithms used in this work can be found at https://github.com/kevinwsbr/vulnfixes

Chapter 2

Background

This chapter presents the main concepts related to information security that serve as a basis for this work and discusses some related work.

2.1 Information Security

According to the ABNT NBR ISO/IEC 17799:2005 standard, information is an essential asset to a company's business and, therefore, needs to be adequately protected. This is especially important in the business environment, which is increasingly interconnected. As a result of this incredible increase in interconnectivity, information is now exposed to an increasing number and a wide variety of threats and vulnerabilities. Information can exist in different forms. It can be printed or written on paper, stored electronically, transmitted by mail or electronic means, presented on film, or spoken in conversations. Regardless of the form presented or the means through which information is shared or stored, it is recommended that it always be adequately protected.

In its introductory section, the ABNT NBR ISO/IEC 17799:2005 standard defines information security as "the protection of information from various types of threats to ensure business continuity, minimize business risk, maximize return on investments and business opportunities." [Sêmola, 2013] defines information security as "an area of knowledge aimed at protecting information and associated assets against unavailability, undue changes, and unauthorized access".

Therefore, when both definitions converge, information security can be defined as the strategies to protect data and information systems from unauthorized access, use, disclosure, interruptions, modifications, or destruction. In this sense, the main objective of information security is to protect information in aspects related to its confidentiality, integrity, and availability.

The CIA triad stands for confidentiality, integrity, and availability and is a foundational model for framing information security policies within organizations. Sometimes

termed the AIC triad to distinguish it from the Central Intelligence Agency, this model emphasizes three core cybersecurity principles.

Figure 2.1: Information security attributes: confidentiality, integrity, and availability.

Dividing these critical concepts into distinct focus areas helps security teams identify specific strategies for each. According to [Sêmola, 2013], these concepts are defined as follows:

2.1.1 Confidentiality

Confidentiality is about ensuring that data remains private. This involves controlling access to prevent unauthorized data sharing, both intentional and accidental. It is essential to restrict access to sensitive resources and grant the necessary privileges to the right people. Protecting confidentiality involves neutralizing direct attacks such as man-in-the-middle (MITM) attacks and accidental access resulting from human error or weak security. Access control policies, encryption, and multi-factor authentication (MFA) are vital to ensuring confidentiality.

2.1.2 Integrity

Integrity is about ensuring that data is reliable and authentic. Integrity violations, often deliberate, can involve the bypass of security systems or accidental errors. Protecting integrity involves using techniques such as hashing, encryption, digital certificates, and non-repudiation measures, ensuring that data has not been compromised.

2.1.3 Availability

Availability is about ensuring that data and systems are accessible when necessary. Systems, networks, and applications must function correctly. Availability challenges can arise due to natural disasters, power outages, or cyber-attacks like DoS or ransomware.

Maintaining availability involves using redundant systems, regular software updates, and comprehensive disaster recovery plans.

2.2 Related concepts

Understanding information security beyond the basic concepts already presented also requires understanding secondary concepts related to data protection, such as asset, vulnerability, threats, attacks, and risk. These concepts will be presented in this section.

2.2.1 Asset

An asset is any value element to an organization, be it a human or technological resource. [Sêmola, 2013] defines an asset as every element that makes up the processes that manipulate and process information, including the information itself, the medium in which it is stored and the equipment in which it is handled, transported and discarded.

2.2.2 Vulnerability

Vulnerability is the weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source [Paulsen and Byers, 2019]. According to [Sêmola, 2013], vulnerabilities are weaknesses present in or associated with assets that manipulate and/or process information that, when exploited by threats, allow a security incident to occur, negatively affecting one or more information security principles. Vulnerabilities are flaws that do not cause incidents, as they are passive elements that depend on an agent to exploit them, making them threats to the organization's security.

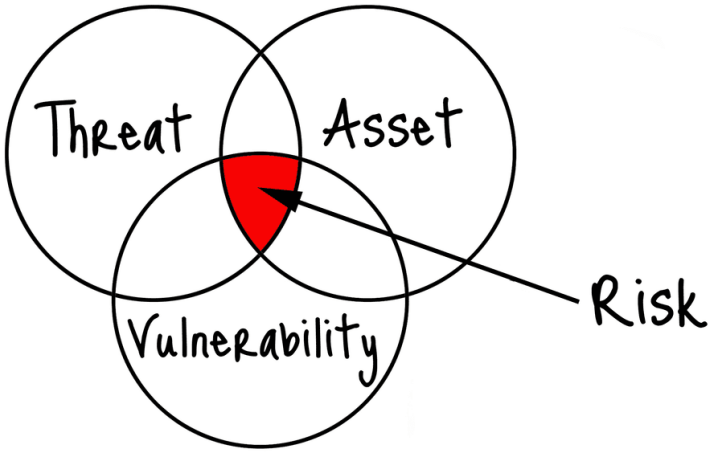

Figure 2.2: The asset, threat, and vulnerability model for information risk.

2.2.3 Threat

A threat can be defined as any event that exploits vulnerabilities and attributes them as the potential cause of an unwanted incident that could damage a system or organization. According to [Cawthra et al., 2020], a threat is any circumstance or event with the potential to adversely impact organizational operations. According to [Sêmola, 2013], threats are agents or conditions that cause incidents that compromise information and its assets through the exploitation of vulnerabilities, causing losses of confidentiality, integrity, and availability and, consequently, causing impacts on a company's business. Threats can be divided into three groups:

- **Natural**: these are those resulting from natural phenomena, such as natural fires, floods, and earthquakes, among others.
- **Involuntary**: unconscious, almost always caused by lack of knowledge, such as accidents, errors, lack of energy, among others.
- **Voluntary**: purposeful, caused by human agents, such as hackers, invaders, spies, and thieves.

2.2.4 Attack

According to [Coelho et al., 2014], an attack is defined as "[...]any action that compromises the security of an organization" and "[...]a deliberate act of attempting to bypass security controls with the objective of exploiting vulnerabilities." According to [Coelho et al., 2014], there are four attack models:

- **Interruption**: an interruption attack occurs when an asset is destroyed or becomes unavailable (or unusable), characterizing an attack against availability.
- **Interception**: an interception attack occurs when an asset is accessed by an unauthorized party (person, program, or computer), characterizing an attack against confidentiality.
- **Modification**: a modification attack occurs when an asset is accessed by an unauthorized party (person, program, or computer) and further altered, characterizing an attack against integrity.
- **Fabrication**: a fabrication attack occurs when an unauthorized party (person, program, or computer) inserts counterfeit objects into an asset, characterizing an attack against authenticity.

[Coelho et al., 2014] also defines two types of attack, passive and active:

- **Passive**: the passive attack is based on listening and monitoring transmissions, with the aim of obtaining information that is being transmitted. Eavesdropping on a telephone conversation is an example of this category. Attacks in this category are difficult to detect because they do not involve data changes; however, they can be prevented with the use of encryption.

- **Active**: active attacks involve modifying data, creating counterfeit objects or denial of service, and have properties opposite to those of passive attacks. These attacks are difficult to prevent due to the need for complete protection of all communication and processing facilities at all times. Therefore, it is possible to detect them and apply measures to recover the losses caused.

2.2.5 Risk

According to the ABNT NBR ISO/IEC 27005:2011 standard, risk in the context of information security is defined as the possibility of a given threat exploiting vulnerabilities in an asset or set of assets, causing negative impacts on the organizatioIn general, the risk to an organization is the intersection of:

- The vulnerabilities and threats to the organization;

- The likelihood that the vulnerability and threat event will be realized;

- The impact to the organization should the event be realized.

2.3 Evaluation metrics

This section describes in detail which metrics were considered to evaluate the source codes obtained.

2.3.1 Complexity

Complexity metrics are used to determine how complex a piece of software is. The most common form of this metric is the Cyclomatic Complexity [McCabe, 1976]. Cyclomatic Complexity measures the number of linearly independent paths through a program's source code. This is done by analyzing its control flow graph. Generally, higher Complexity suggests a code that is more difficult to understand and maintain.

2.3.2 Number of Lines of Code (NLOC)

NLOC refers to the count of lines in a computer program that are not comments or blank lines, essentially representing the lines that contribute to the actual codebase. It's

a simple measure of software size and is used to assess the amount of code in a software project.

2.3.3 Token Count

The token count is a measure of the number of individual pieces of syntax in the source code. Tokens are the smallest elements of a program, such as keywords, operators, identifiers, and symbols. Counting tokens is a way to measure the size and Complexity of code, but in a way that is more detailed than simply counting lines of code.

2.3.4 Delta Maintainability Model (DMM)

The Delta Maintainability Model (DMM) is an innovative approach to evaluating the maintainability of code changes in a fine-grained manner. The DMM measures the maintainability of a code change as the ratio between low-risk code and the overall code modified. The DMM identifies code riskiness by reusing software metrics and risk profiles of the SIG Maintainability Model (SIG-MM) while applying new aggregations and scoring for software metric deltas at the level of fine-grained code changes like commits or pull-requests instead of aggregating at the system level [di Biase et al., 2019].

The core concept of DMM is to calculate the maintainability proportion of code changes (additions or removals of lines of code, LOC) based on the risk profiles of the units and modules affected by these changes. It uses five key code properties: Duplication, Unit Size, Unit Complexity, Unit Interfacing, and Module Coupling, each with defined low-risk criteria. DMM measures and classifies code changes into each property's risk profile (low, medium, high, very high). In this work, the following DMM metrics were considered:

DMM Unit Size: The Unit Size metric refers to the size of the source code units, determined by the number of lines of code (NLOC), excluding lines of only whitespace or comments. A unit with 15 LOC or less is considered low-risk. This metric is crucial because smaller, more concise units of code are generally easier to understand and maintain.

DMM Unit Complexity: The Unit Complexity metric assesses the complexity within the smallest executable parts of the source code, such as methods or functions. Complexity is gauged using McCabe's cyclomatic complexity. A unit with a McCabe complexity of 5 or less is deemed low-risk. This metric is significant as it helps to understand the intricacy of code changes and their impact on maintainability.

DMM Unit Interfacing: The Unit Interfacing metric in DMM is about the size of the interfaces of the units in terms of the number of interface parameter declarations. Units with at most two parameters are categorized as low-risk, indicating that simpler interfaces typically lead to better maintainability. This aspect is crucial for evaluating how changes in the interfaces of code units affect the overall maintainability.

2.4 Related works

Several previous works have proposed the creation of datasets that search for vulnerabilities associated with code snippets. In [Bhandari et al., 2021], the authors proposed a dataset covering 5,365 CVE records for 1,754 open-source projects that were addressed in a total of 5,495 vulnerability fixing commits without performing any temporal analysis or categorization by language or group of vulnerabilities. In [Fan et al., 2020], the presented dataset contains only vulnerabilities found in applications in C/C++ languages. In [Challande et al., 2022], the authors extracted and analyzed vulnerabilities from one project: the Android Open Source Project (AOSP). In these three works, the number of vulnerabilities collected, languages analyzed, and projects cataloged is lower than that of the dataset presented in this work.

In [Pianco et al., 2016], the authors explore the impact of the change history of functions on the existence of vulnerabilities in software. By analyzing over 95,000 functions from the Mozilla and Linux Kernel projects, the study investigates how the frequency of function changes can indicate their vulnerability to security threats. [Alves et al., 2016a] evaluates various machine learning techniques for predicting vulnerabilities in software using a comprehensive dataset encompassing 2,186 vulnerabilities from five open-source projects. It reveals that while specific techniques can predict nearly all vulnerabilities in the dataset, they do so with low precision. Together, [Pianco et al., 2016] and [Alves et al., 2016a] bring vulnerabilities from just five projects, a considerably smaller number than proposed in the dataset presented in this work.

In [Alves et al., 2016b], the authors investigate the relationship between software metrics and security vulnerabilities. Using a dataset built from 2,875 security patches across five widely-used projects, the study investigates whether software metrics can reflect the characteristics leading to vulnerabilities. The analysis shows that while software metrics can distinguish between vulnerable and non-vulnerable functions, finding strong correlations between these metrics and the number of vulnerabilities proved challenging. As in the [Alves et al., 2016a], this work catalogs vulnerabilities in only five different projects.

In the software vulnerability life cycle context, previous work has been done to analyze the evolution of different vulnerabilities to your discovery, patching, and exploitation. In [Shahzad et al., 2012], there was a more extensive and in-depth analysis of the entire lifecycle of a software vulnerability, from the moment of its discovery to its publication, patching, and exploitation. However, this study only covers vulnerabilities published up to 2011, and the data sources used for the proposed analyses are no longer publicly available, which prevents an analysis of this work with more up-to-date data. In contrast, the dataset presented in this work covers vulnerabilities published over a longer period, from 2009 to 2022.

Chapter 3

Methodology

This chapter presents the methodology adopted in this work to define the structure of the dataset, development of the tools used to collect and process the data and the research questions selected to validate the dataset presented.

3.1 Dataset construction

This section will detail the steps taken to construct the dataset, including the source selection process and the structure of the dataset.

3.1.1 Data sources

The proposed dataset was created using two main sources: the National Vulnerability Database (NVD), which is a collection of vulnerability management data maintained by National Institute of Standards and Technology (NIST), and GitHub, a platform used for hosting source code. These sources offer various real-life instances of software vulnerabilities and their respective fixes.

National Vulnerability Database

The National Vulnerability Database (NVD) is an extensive repository that provides information about security vulnerabilities in software. The NIST maintains it and serves as a database containing all Common Vulnerabilities and Exposures (CVE) records. Each vulnerability record published in NVD has a standardized structure, which includes:

- Vulnerability description;
- Affected software versions;
- Severity and impact metrics;
- References.

NVD was chosen for CVE extraction in this project due to its comprehensive and reliable nature. As a primary source for CVE data, NVD provides a well-structured and up-to-date repository of vulnerabilities, each with a unique identifier and detailed metadata. Using NVD ensures that data extracted from CVE is reliable and covers various software and vulnerabilities. This is particularly important for the objectives proposed in this work, which consist of analyzing vulnerabilities in a wide spectrum of software systems and programming languages.

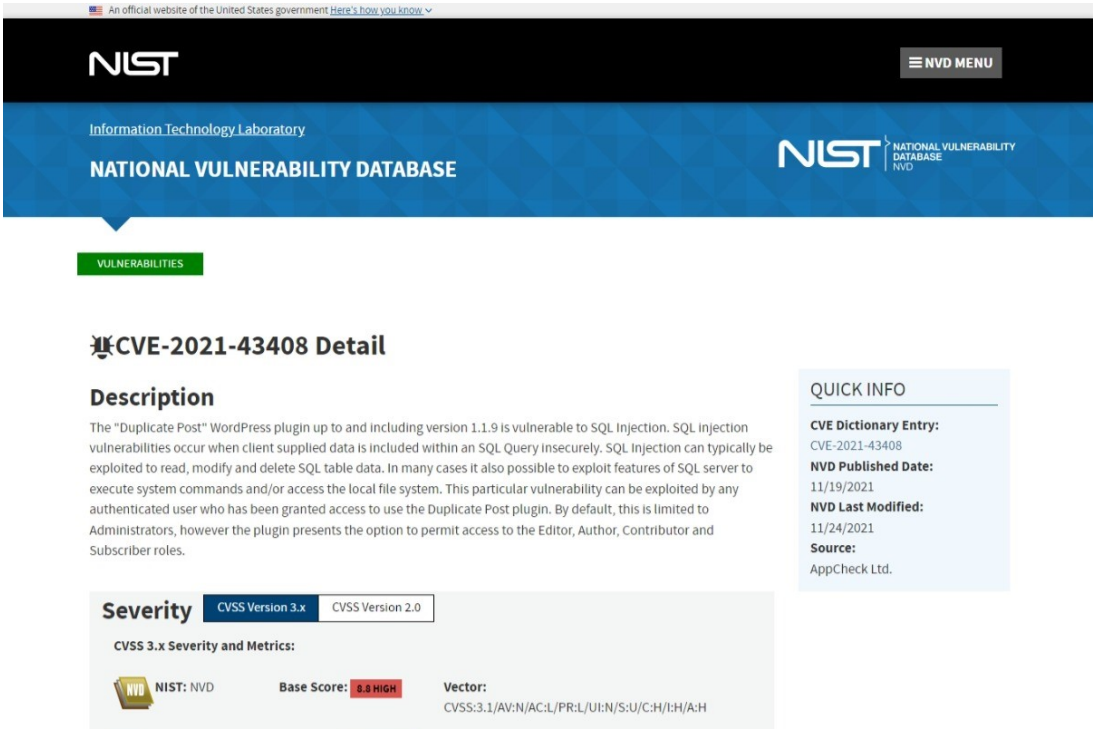

Figure 3.1: NVD page with the SQL Injection vulnerability record in a WordPress plugin.

The NIST performs an exhaustive analysis of CVEs published in the CVE Dictionary. The NVD staff analyzes CVEs by aggregating data points from the description, references supplied, and any supplemental data that can be found publicly at the time. This analysis results in the association of impact metrics (Common Vulnerability Scoring System - CVSS), vulnerability types (Common Weakness Enumeration - CWE), and applicability statements (Common Platform Enumeration - CPE), as well as other pertinent metadata.

The CVSS is a standardized framework for assessing and quantifying software vulnerability severity. CVSS offers a numerical score that helps organizations and security professionals understand a vulnerability's potential impact and risk. Three primary versions of CVSS are available: $CVSS_2$, $CVSS_3$, and $CVSS_{3.1}$. $CVSS_2$, introduced in 2007, considers different factors, including access complexity, authentication requirements, and impact metrics, to assign a score ranging from 0 to 10, where 10 indicates the most severe vulnerability. It aimed to provide a systematic method of prioritizing security patches and comprehending the potential consequences of a vulnerability.

In contrast, $CVSS_3$, released in 2015, brought several notable changes, such as a more detailed approach to base, temporal, and environmental metrics. $CVSS_{3.1}$, a subsequent update, introduced supplementary metrics to incorporate nuanced aspects of vulnerabilities, including their effects on confidentiality, integrity, and availability, further refining the scoring system. The overall precision of scores increased significantly with the update, resulting in an even higher level of usefulness for organizations in determining vulnerability remediation and mitigation strategies.

GitHub

GitHub is a widely used platform for version control and collaborative software development using Git. GitHub acts as a code hosting service, allowing developers to store, manage, and track changes to the source code of developed applications. GitHub's extensive use in the software development community means it hosts vast open-source code, including projects of significant size and complexity across different programming languages. Additionally, GitHub's API facilitates automated data extraction, allowing information about repositories, commits, and code changes to be extracted in a streamlined way. These factors were decisive for choosing GitHub as a source for obtaining source codes for the reported vulnerabilities.

3.1.2 Dataset structure

Figure 3.2 shows the proposed dataset diagram. The tables were structured as follows:

- **CVEs**: represents the CVEs published by NVD. Each record is identified by an ID, its description, the published and last modified dates of the record in the NVD, the CVSS risk, impact and exploitability assessment metrics, and information about the attack vectors and impact ratings on confidentiality, integrity, and availability of the affected service;
- **CWEs**: represents the CWEs published by MITRE. Each record is identified by an ID, its short and long descriptions, and a link to its web page;
- **Classifications**: represents the associations between CVEs and CWES;
- **Fixes**: represents the fixes applied to resolve vulnerabilities. Each record is linked to vulnerabilities and commits, indicating the code changes that fixed that vulnerability;
- **Commits**: represents the collected fix commits. Each record includes the commit message, author, publish date, and information about the change size and complexity metrics of the changes made;

- **Files**: represents the files in which vulnerabilities were found and fixed. Each record contains the file name, language, the paths before and after the change, the differences introduced by the commit (diff), and various code complexity metrics;
- **Repositories**: represents the code repositories in which vulnerabilities were found. Each record contains the repository's URL, name, responsible user, creation and update date, stars and forks count, and the primary language;
- **Methods**: represents the modified methods in the source code, indicating where the vulnerabilities were present and where the fixes were applied. Each record contains information about the method name, parameters, file position, and associated complexity metrics.

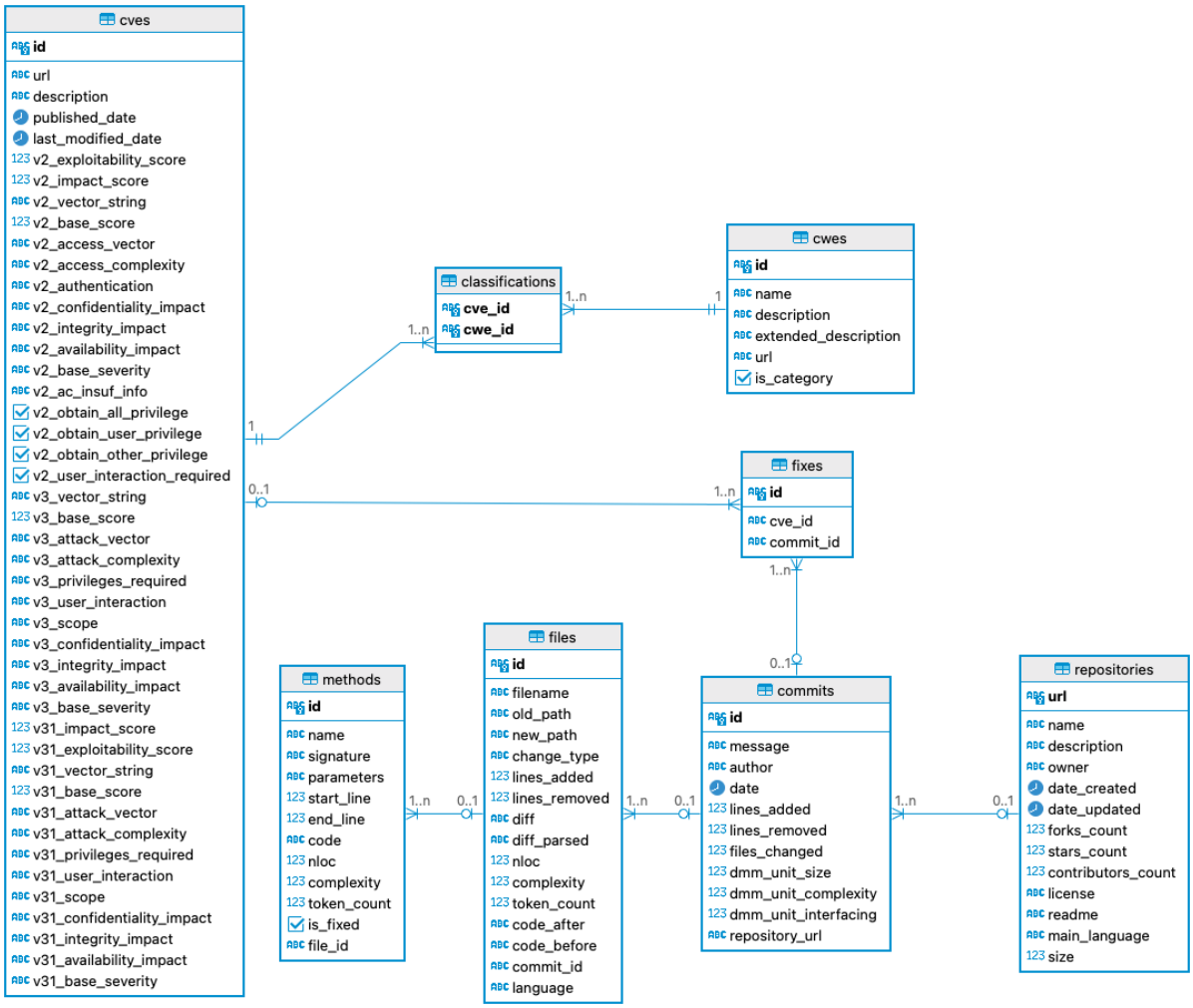

Figure 3.2: Entity-Relationship Diagram of the proposed dataset.

3.2 Development of the extraction tool

Figure 3.3 illustrates the detailed diagram of the dataset construction process proposed in this work. In phase 1, vulnerabilities published by NVD since 1999 were collected and filtered. In phase 2, the associated CWEs were identified and extracted to categorize published vulnerabilities. In phase 3, the repositories and fix commits related to each CVE were extracted. In phases 4 and 5, the collected repositories and fix commits were analyzed, which made it possible to identify the changed files and methods.

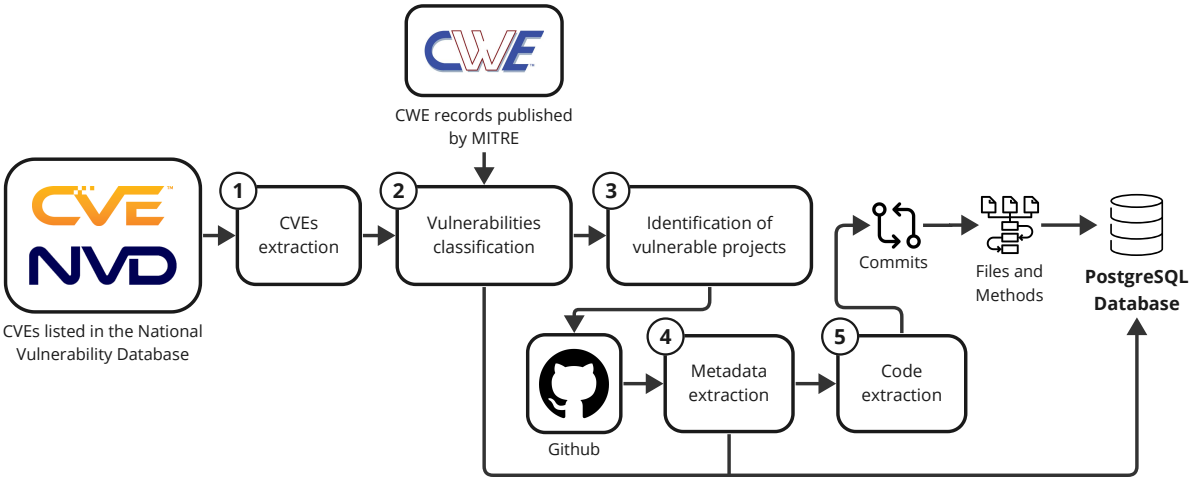

Figure 3.3: Dataset construction flow.

The tools used for the collected data's extraction, selection, and pre-processing steps were developed using the Python language in version 3.11.2, and the relational database management system (RDBMS) chosen to store the data was PostgreSQL in version 14.2. Each phase of the dataset construction process has been detailed in the following subsections:

3.2.1 CVEs extraction

A tool was developed to automate the collection of information related to previously cataloged software vulnerabilities. For this, the tool uses the NVD API to download all the CVEs published since 1999, the date of the first publication, until July 2023. Each CVE record is returned in JSON format and provides crucial details such as the CVE-ID, publication date, description, reference links, and metrics that classify the severity of the vulnerability in question.

The automatic extraction process collected 226,017 CVEs. However, when carrying out a detailed analysis of this data, it became evident that many records did not contain sufficient information to answer the questions proposed in this study. Therefore, a filtering step became necessary to identify CVEs that met the following desired criteria:

- Status: ”Published”;
- One or more CWEs associated;
- CVSS metrics available in versions 2, 3.0 or 3.1;
- One or more references pointing to repositories hosted on GitHub containing the code with the fix for the presented vulnerability.

During this process, the database was significantly refined, reducing the number of CVEs from 226,017 to 11,558, resulting in a more accurate and comprehensive dataset that provides essential insight into the nature and severity of each vulnerability and access to associated code snippets.

3.2.2 Vulnerabilities classification

The Common Weakness Enumeration (CWE) is a community-developed list of standard software and hardware weakness types with security ramifications. A ”weakness” is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to introducing vulnerabilities. The CWE List and associated classification taxonomy serve as a language that can be used to identify and describe these weaknesses in terms of CWEs, along with information about how they can be exploited and recommendations for mitigating or fixing the issue.

Each CVE collected has one or more associated CWEs, denoted by the code ”CWE-<ID>”. In addition, NVD uses two supplemental classifications, ”NVD-CWE-noinfo” and ”NVD-CWE-other”, to indicate cases where there is insufficient detail to classify a vulnerability or where the NVD classification does not match an equivalent CWE.

3.2.3 Identification of vulnerable projects

Vulnerable projects were identified by analyzing the URLs labelled as fixes in the ”references” section of the CVEs collected. Regular expressions were used to pinpoint links to published GitHub fix commits for each existing entry. Each commit is assigned a unique identifying code called a ”hash”. This hash allowed us to determine the URL of the corresponding repository and the specific project to which the code belongs.

3.2.4 Metadata extraction

For each repository collected, the GitHub API was used to extract the metadata associated with each project. During this phase, it was observed that some previously fetched commits became unavailable because their repositories had been deleted or were no longer

available. To solve this, the repositories were filtered to keep only those that were accessible and had source code available.

The data obtained from the resultant repositories includes the name of the repository, description, date of creation, last date of push, number of collaborators involved, number of forks, number of stars and the license under which the code is made available. This information provides significant insights into the maturity and complexity of the projects and the community involved in the development process, allowing for a more comprehensive understanding of the relevance and context of each project.

3.2.5 Code extraction

The previously identified hashes, obtained through the extraction of vulnerable projects via the CVE references, indicate the commit where the vulnerability was addressed. With this, it was possible to identify which code versions were released before and after the fix's implementation.

For each commit, it was extracted the associated metadata, revealing information about the committer, the associated message, the commit date, a log of the files modified, and the delta maintainability model measurements associated with the change. Similarly, information was obtained about the changed files, such as name, programming language, type of the modification(i.e., added, deleted, modified, or renamed), number of lines added or removed, and the complexity of the changes introduced. In addition, more specific information about the modifications made to the existing methods was extracted using the *PyDriller* tool, such as their name, signature, parameters, modified code, and the modification's complexity.

3.3 Data analysis

The dataset presented in this work contains information about vulnerabilities published in NVD about several open-source projects and their fixes. With the data collected, it was possible to analyze the evolution of vulnerabilities reported over the years and check how they behave in different programming languages. To this end and aiming to validate the presented dataset and its importance, the following research questions guided the analysis process:

RQ1 - How does the incidence of reported software vulnerability compare for different programming languages?

The main objective of this research question was to analyze project repositories that have published CVEs to identify which languages were used in the development of these projects. With this, it was possible to obtain a more in-depth understanding of the

types of vulnerabilities that occur recurrently in the most popular programming languages currently in use.

RQ2 - What are the characteristics of vulnerability fixes?

This research question evaluates the code complexity of the solutions implemented to resolve the reported vulnerabilities. To achieve this, the components of the modified code snippets, such as files, methods, and changed lines of code, were analyzed to extract complexity metrics.

Chapter 4

Results

This chapter presents the results obtained in this work, including the characteristics presented by the constructed dataset and the results obtained for the research questions.

4.1 Dataset summary

Table 4.1 presents a summary of the proposed dataset. 11,558 CVEs were collected from a total of 3,587 open-source software projects developed in 58 different programming languages (PL). 11,332 fix commits for these vulnerabilities were identified, with changes made to 23,360 files and 99,182 unique methods. CVEs were classified into 316 distinct CWE types. The number of commits is lower than the number of CVEs because some commits were responsible for resolving more than one CVE simultaneously.

Table 4.1: Summary of the presented dataset.

CVEs	**CWEs**	**Projects**	**Programming languages**	**Commits**	**Files**	**Methods**
11,558	316	3,587	58	11,332	23,360	99,182

Table 4.2 presents a summary of security vulnerabilities across various programming languages quantified by CVE and CWE counts. The ten programming languages that present the highest number of reported CVEs were selected to make the data presentation more straightforward. This choice was made to optimize the understanding of the dataset, focusing on the languages that demonstrate a higher incidence of security vulnerabilities and thus offering a more targeted analysis for the most critical programming languages in terms of reported vulnerabilities. Language C tops the list with a substantial lead, having 2,566 CVEs reported and 135 different CWEs identified. This high number reflects the widespread use of C in various systems and its legacy codebase, which often contributes to a higher number of discovered vulnerabilities. PHP follows with 1,046 CVEs and 110 CWEs, indicating a significant number of security issues that could be due to its extensive use in web applications, which are common targets for security attacks.

Table 4.2: Summary of the top 10 programming languages with most CVEs.

Programming language	**CVEs**	**CWEs**	**Commits**	**Contributors**
C	2,566	135	2,620	52,672
PHP	1,046	110	1,024	20,662
C++	726	100	750	16,331
JavaScript	568	107	561	22,843
Python	457	109	507	29,028
Java	348	107	379	13,690
Go	331	100	314	19,697
Ruby	246	68	268	12,239
TypeScript	169	53	179	12,774
Vim Script	166	20	168	722

Furthermore, the table demonstrates that newer languages like TypeScript and Go have lower CVE counts of 169 and 331, respectively. However, their range of common weaknesses is also less diverse. This may indicate that newer languages can incorporate more robust security features.

Table 4.3: Summary of the projects with most CVEs.

Project	**CVEs**	**CWEs**	**Contributors**	**Stars**	**Forks**	**PL**	**Commits**	**Files**	**Methods**
linux	1,084	71	14,678	50,846	159,752	C	1,110	1,554	4,326
tensorflow	399	39	410	89,056	178,469	C++	412	794	2,180
vim	166	20	280	5,084	33,281	Vim Script	167	204	480
gpac	118	23	63	486	2,355	C	113	139	342
tcpdump	111	8	156	817	2,365	C	114	137	397
pimcore	72	9	286	1,357	3,030	PHP	64	108	300
xwiki-platform	70	29	185	501	878	Java	78	252	1,022
radare2	63	19	356	2,915	18,779	C	60	84	294
discourse	58	23	383	8,150	38,972	Ruby	66	214	732
php-src	51	18	236	7,677	36,168	C	49	54	230

Table 4.3 presents a comprehensive summary of the top ten projects with more CVEs reported, detailing key metrics that reflect their development and community engagement. The table lists projects such as Linux, TensorFlow, and Vim. This data highlights the popularity and scale of these projects and provides a snapshot of their current state in terms of security, community engagement, and development activity.

Table 4.4 presents a general comparison between the characteristics of the main datasets found in the literature and the dataset presented in this work.

Table 4.4: Comparative table between the characteristics of the main vulnerability datasets extracted from real projects present in the literature.

Dataset	CVEs	CWEs	Projects	Languages	Commits	Files	Methods
[Alves et al., 2016a]	-	-	5	2	5,750	15,490	10,385
[Pianco et al., 2016]	-	-	2	2	1,010	-	1,784
[Fan et al., 2020]	3,754	91	348	-	4,432	8,143	11,823
[Bhandari et al., 2021]	5.365	180	1,754	31	5,495	18,249	50,322
[Nikitopoulos et al., 2021]	5,131	168	1,675	48	5,877	13,738	-
[Challande et al., 2022]	1,991	-	1,800	3	1,275	-	-
This work	**11,558**	**316**	**3,587**	**58**	**11,332**	**23,360**	**99,182**

4.2 Data analysis

The data extracted from the dataset presented in this work and displayed in the tables 4.1, 4.4 and 4.3 listed above and in the figures 4.1, 4.2, 4.3, 4.4 and 4.5, allows it to be possible to answer the following RQ:

RQ1 - How does the incidence of reported software vulnerability compare for different programming languages?

Figure 4.1 shows the percentage of vulnerabilities with associated fix commits for each programming language. It can be seen that the C language leads with the highest percentage of reported vulnerabilities with 28.72%, followed by PHP with 16.76% and other languages with lower percentages.

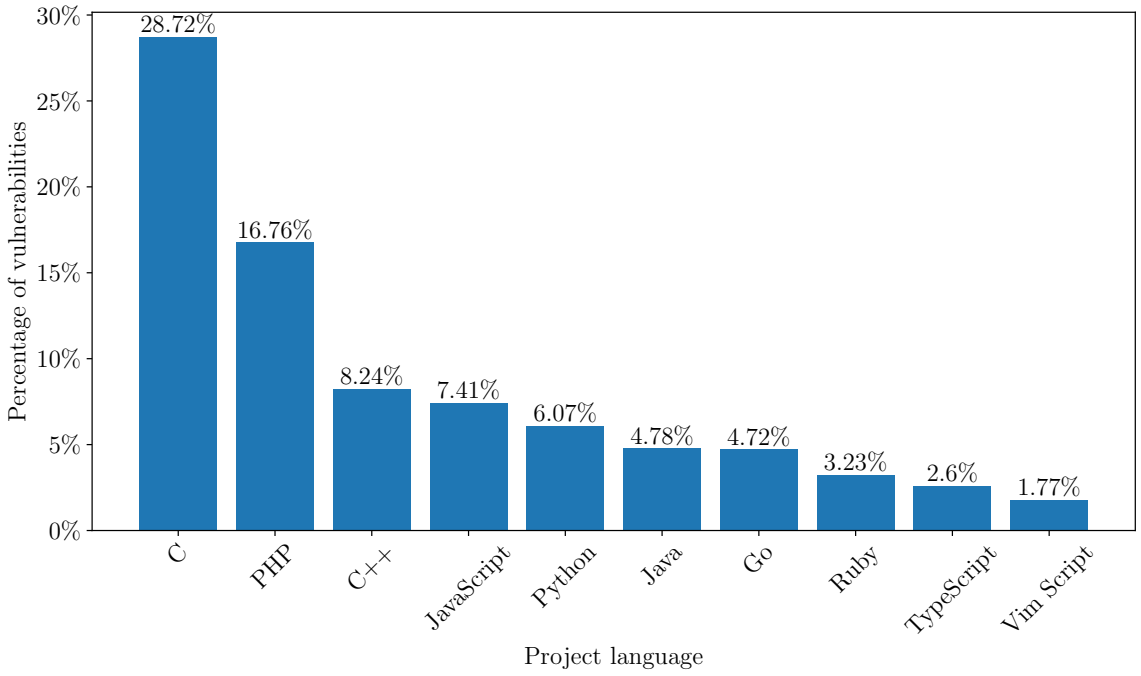

Figure 4.1: Percentage of vulnerabilities by programming languages.

Figure 4.2 shows the proportion of contributors in different programming languages. The C language has the highest percentage of contributors, with 23.86%, followed by Python with 12.03% and JavaScript with 10.93%. The other languages, such as PHP, Go, C++, Java, Ruby, TypeScript, and Rust, have progressively smaller percentages of contributors.

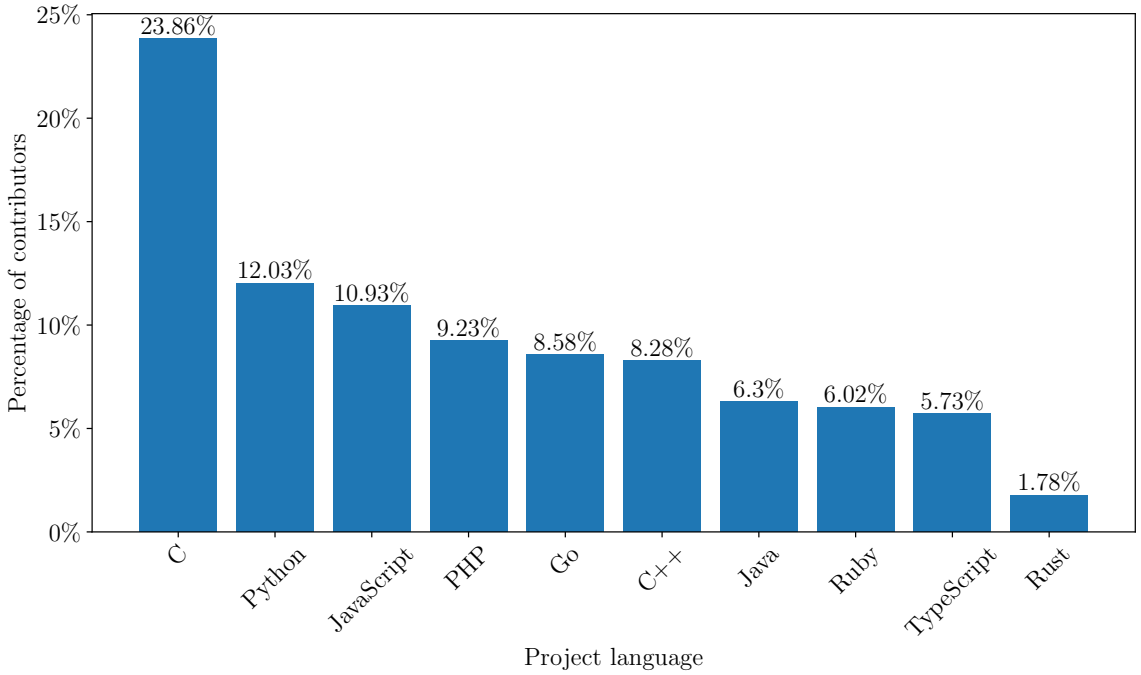

Figure 4.2: Percentage of contributors by projects languages.

C simultaneously has the highest number of contributors and the highest percentage of vulnerabilities, suggesting that popularity and widespread use may be correlated with a more significant number of known vulnerabilities. On the other hand, languages like Python and JavaScript, although they are in the top three in terms of contributors, have a lower percentage of reported vulnerabilities compared to the C language. It can also be seen that languages like TypeScript and Rust have fewer reported vulnerabilities. Fewer vulnerabilities could result from a more secure design or a smaller user base, leading to fewer discovered vulnerabilities.

Figure 4.3 presents an overview of the number of vulnerabilities with associated fix commits presented by the ten programming languages with the most reported vulnerabilities between 2009 and 2022. After applying the previously mentioned filters to select the CVEs, finding commits related to vulnerabilities reported before 2009 was impossible. Therefore, records from previous years are not present in this temporal analysis.

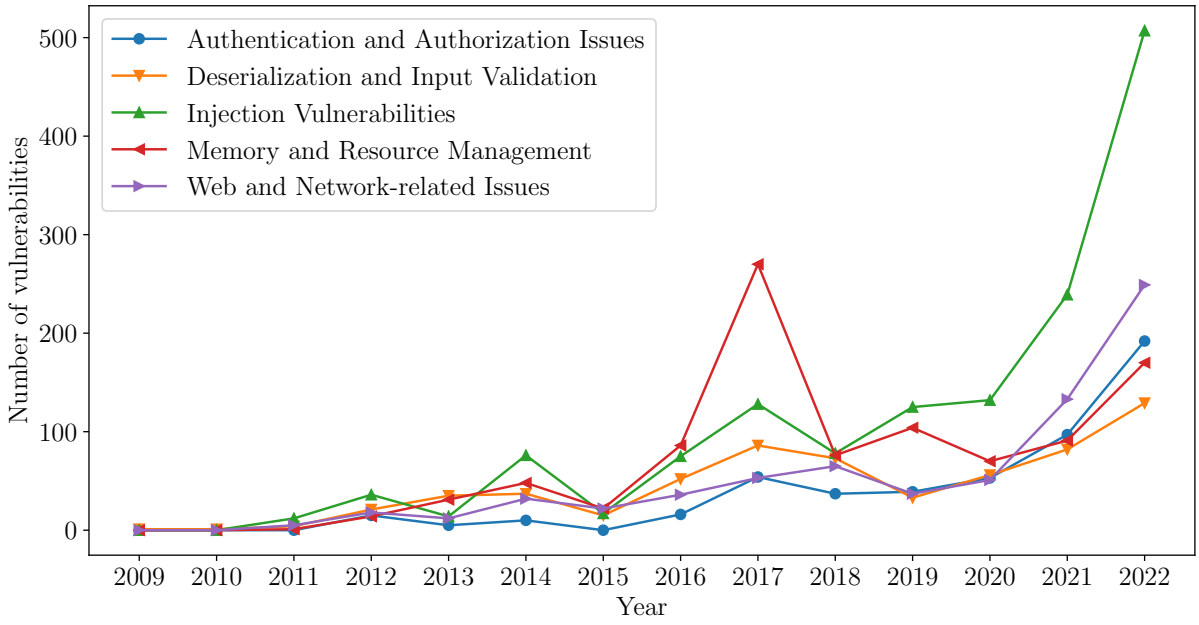

Figure 4.3: Number of vulnerabilities reported over the years.

The general trend shown by the data indicates an increase in the number of vulnerabilities in almost all of the programming languages in the top 10, which may be due to an increase in the use of these languages, an increase in the complexity of the software being developed, or more accurate methods of vulnerability detection. This growing trend highlights the ongoing challenges in software security and the importance of rigorous security practices in software development and maintenance.

According to CVE Details [MITRE, 2023a], more than 14600 vulnerabilities were reported in 2017, compared to the 6447 reported in 2016. This historic peak can be seen when analyzing the same behavior presented by the C language, with a significant increase in reported and fixed vulnerabilities in 2017.

The CWE Top 25 Most Dangerous Software Weaknesses List [MITRE, 2023b] is a free, easy-to-use community resource that identifies the most widespread and critical programming errors that can lead to serious software vulnerabilities. These weaknesses are often easy to find, and easy to exploit. They are dangerous because they are often easy to find, exploit, and allow adversaries to completely take over a system, steal data, or prevent an application from working.

To facilitate the analysis process, the CWE Top 25 Most Dangerous Software Weaknesses List was used as a basis to group all vulnerabilities present in the proposed data set into five subgroups, organized as follows:

Group 1: Injection Vulnerabilities: vulnerabilities in this group occur when an attacker sends malicious data to the application as part of a command or query. Such data may cause the application to execute unintended commands or access unauthorized data.

- CWE-787: Out-of-bounds Write
- CWE-79: Cross-site Scripting (XSS)
- CWE-89: SQL Injection
- CWE-78: OS Command Injection
- CWE-77: Command Injection

Group 2: Memory and Resource Management: vulnerabilities in this group are related to the mismanagement of memory and other critical resources in the application, which can lead to crashes, performance issues, or security vulnerabilities.

- CWE-416: Use After Free
- CWE-125: Out-of-bounds Read
- CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer

Group 3: Authentication and Authorization Issues: vulnerabilities in this group are related to weaknesses in verifying a user's identity, process, or application (authentication) and determining whether a requester has the right to access a resource (authorization).

- CWE-287: Improper Authentication
- CWE-862: Missing Authorization
- CWE-476: NULL Pointer Dereference
- CWE-306: Missing Authentication for Critical Function
- CWE-863: Incorrect Authorization
- CWE-276: Incorrect Default Permissions

Group 4: Deserialization and Input Validation: vulnerabilities in this group are related to mishandling of serialized data or poor input validation that may result in application compromise.

- CWE-502: Deserialization of Untrusted Data
- CWE-190: Integer Overflow or Wraparound
- CWE-20: Improper Input Validation

Group 5: Web and Network-related Issues: vulnerabilities in this group are related to web and network security issues, from the security of web sessions to the integrity of communication between applications.

- CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- CWE-352: Cross-Site Request Forgery (CSRF)
- CWE-434: Unrestricted Upload of File with Dangerous Type
- CWE-798: Use of Hard-coded Credentials
- CWE-918: Server-Side Request Forgery (SSRF)
- CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
- CWE-269: Improper Privilege Management
- CWE-94: Improper Control of Generation of Code ('Code Injection')

Figures 4.4 and 4.5 show the distribution of vulnerabilities present in each group in relation to programming languages and the evolution of the number of vulnerabilities per group over the years.

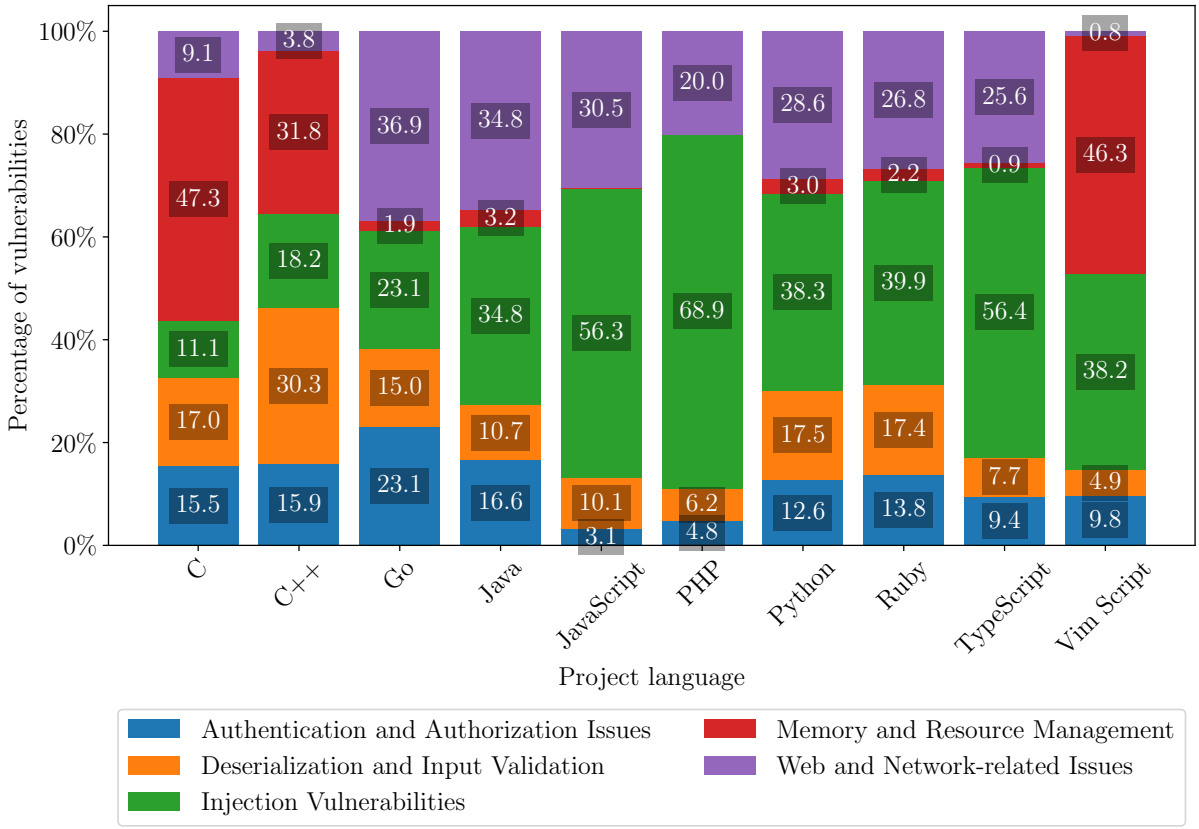

Figure 4.4: Distribution of clusters by languages.

Notably, PHP has a high percentage of vulnerabilities related to Web and Network-related Issues, reflecting its widespread use in web development. Conversely, C and C++ show a significant proportion of their vulnerabilities in Memory and Resource Management, which aligns with the common challenges of managing low-level operations in these languages. Injection vulnerabilities are less dominant in languages such as JavaScript and TypeScript but still represent a notable concern.

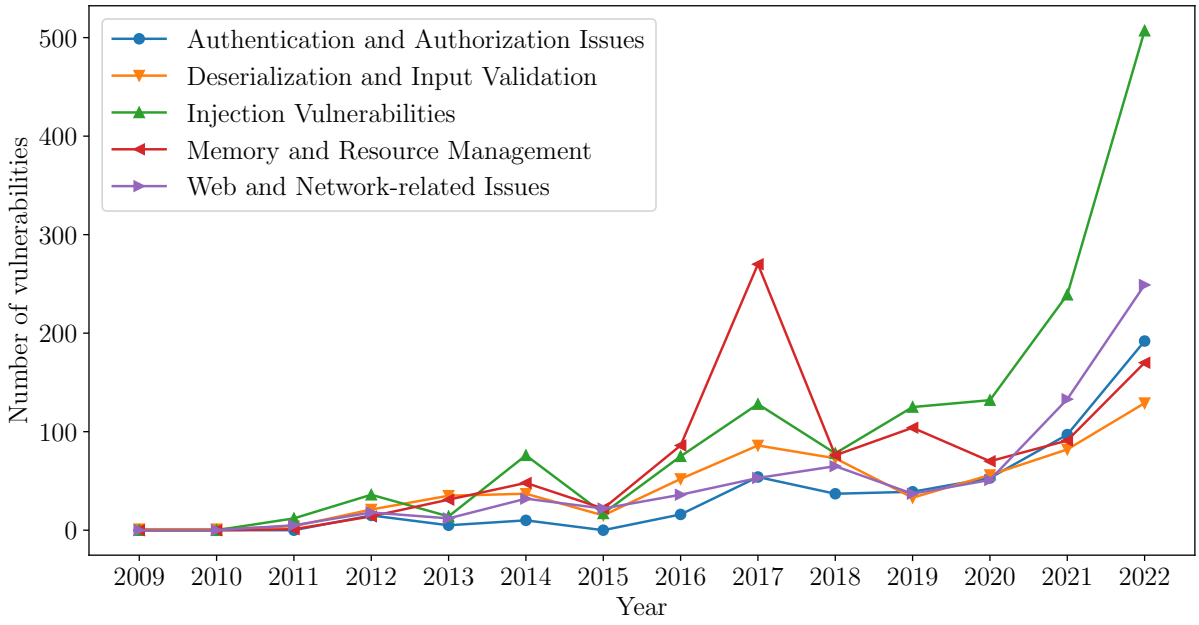

Figure 4.5: Number of vulnerabilities reported per group over the years.

It was possible to extract the tables and figures presented below through the tables of commits, files, and methods existing in the dataset presented. As a result, the data obtained made it possible to answer the following RQ:

RQ2 - What are the characteristics of vulnerability fixes?

Tables 4.5, 4.6 and 4.7 summarize the base scores for software vulnerabilities across different programming languages evaluated by the Common Vulnerability Scoring System (CVSS) versions 2, 3, and 3.1.

Table 4.5: Summary of $CVSS_2$ base scores by programming languages.

Programming language	**Min**	**Max**	**Mean**	**Median**
C	2.1	7.8	5.63	5.8
C++	5	7.5	6.43	6.8
Go	2.6	10	5.36	5.5
Java	3.5	7.5	5.46	5.2
JavaScript	2.1	10	5.21	5
PHP	2.1	9	4.51	4.3
Python	3.5	10	5.69	5
Ruby	3.5	5.2	4.23	4.3
TypeScript	3.5	10	5.47	5
Vim Script	4.3	9.3	6.29	6.8

In the $CVSS_2$ version, the scores are generally lower, with the C programming language having the broadest range of scores and PHP presenting the lowest mean score.

Table 4.6: Summary of $CVSS_3$ base scores by programming languages.

Programming language	**Min**	**Max**	**Mean**	**Median**
C	2.5	9.6	6.76	7.10
C++	7.4	8.8	7.90	7.50
Go	3.5	10	7.24	7.50
Java	4.2	9.8	7.22	7.30
JavaScript	3.5	10	7.33	7.50
PHP	2.6	9.9	6.48	6.50
Python	2.4	10	7.40	7.65
Ruby	3.5	8.1	6.61	7.30
TypeScript	4.3	9.8	7.16	7.50
Vim Script	5.5	8.6	7.41	7.80

In $CVSS_3$, the base scores across all languages have increased. Notably, C++ shows a higher mean in $CVSS_3$ than $CVSS_2$, suggesting an increase in the severity of vulnerabilities identified within projects using this language.

Table 4.7: Summary of $CVSS_{3.1}$ base scores by programming languages.

Programming language	**Min**	**Max**	**Mean**	**Median**
C	3.3	9.8	7.34	7.5
C++	7.5	9.8	8.36	7.8
Go	5.3	9.8	7.68	7.8
Java	4.2	9.8	7.16	6.1
JavaScript	3.3	9.9	6.95	6.5
PHP	3.5	9.8	6.28	6.1
Python	4.3	9.8	7.68	7.5
Ruby	5.4	9.8	6.55	6.1
TypeScript	4.3	9.8	6.84	7.5
Vim Script	3.3	9.8	7.47	7.8

The minimum scores in $CVSS_{3.1}$ consistently increase, with most languages exhibiting higher mean scores than the previous version. C++ and Python show particularly significant increases. The rise in scores may stem from better vulnerability reporting, increasingly complex software systems, and an increased awareness of security implications. This progression across CVSS versions highlights an ongoing maturation in the

vulnerability assessment process and suggests that reported vulnerabilities are becoming more severe and complex.

Figures 4.6, 4.7 and 4.8 presents violin plots that show the distribution of DMM metrics [di Biase et al., 2019] of all commits that have vulnerability fixes that were extracted according to the programming languages that have the most associated CVEs. These metrics show how project maintenance has been affected by security fixes.

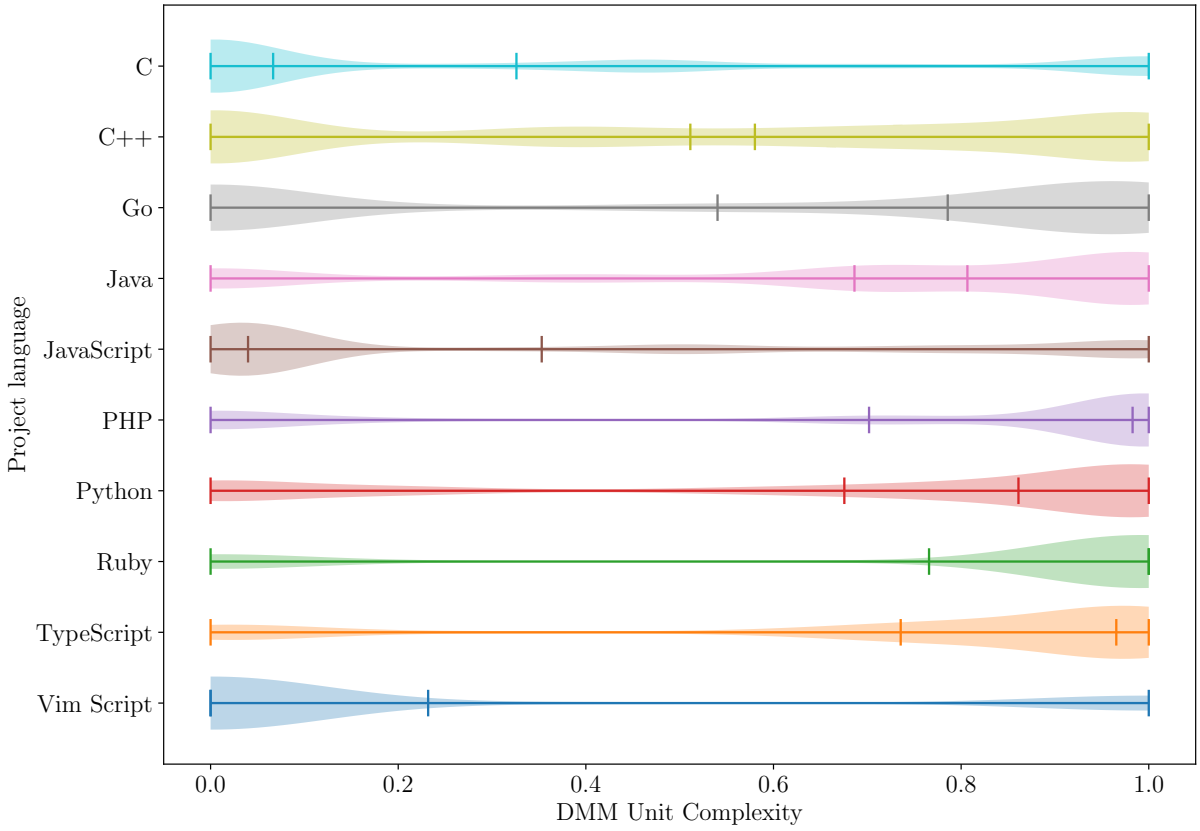

Figure 4.6: DMM unit complexity by programming languages.

Figure 4.6 shows the distribution of method changes depending on their cyclomatic complexity. Languages such as Ruby and TypeScript had a higher proportion of low-risk changes in terms of complexity, meaning that changes made in these languages are more likely to involve less complex methods and, therefore, easier to maintain. On the other hand, C and JavaScript showed more significant variations, indicating more risk in the changes made.

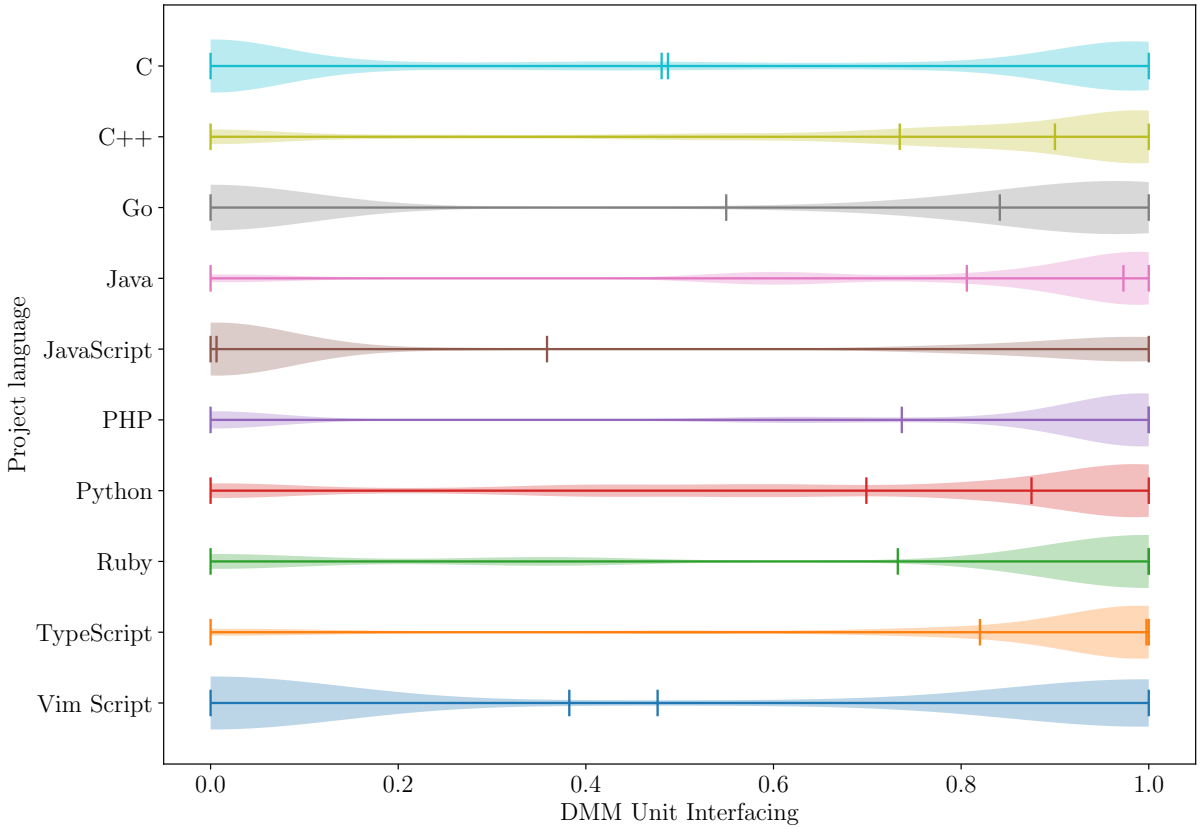

Figure 4.7: DMM unit interfacing by programming languages.

Figure 4.7 shows the changes in the number of method parameters in each language. Languages such as PHP, Python, and Ruby show a higher proportion of low-risk changes, suggesting that methods in these languages are modified to have fewer parameters, potentially making them easier to understand and use.

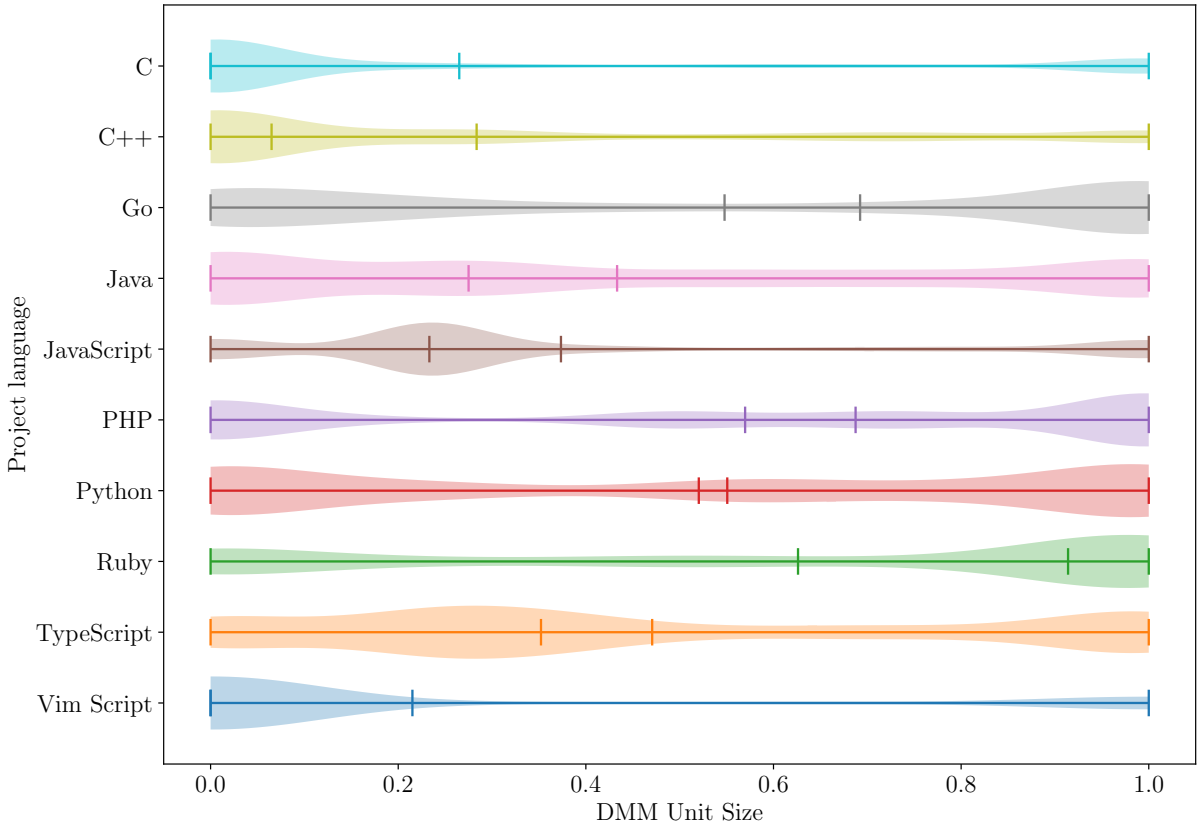

Figure 4.8: DMM unit size by programming languages.

Figure 4.8 shows the changes in the length of the methods in terms of lines of code. Vim Script has a low proportion of minor changes, indicating that many methods were increased while fixing vulnerabilities, which can make them more difficult to maintain. On the other hand, languages like TypeScript and Ruby had a higher proportion of minor changes, suggesting that methods in these languages tend to be smaller and easier to maintain.

Tables 4.8, 4.9 and 4.10 provides a comprehensive view of the nature and scale of vulnerability fixes across various programming languages, highlighting differences in fix sizes, complexity, and syntactic verbosity, focusing on lines of code (NLOC), complexity, and token count.

Table 4.8: NLOC of the changed files by programming languages.

Programming language	**Min**	**Max**	**Mean**	**Median**
C	0	24963	1247.70	679
C++	0	15237	731.40	305
Go	1	45725	659.32	197
Java	1	5356	372.77	148
JavaScript	0	19630	916.83	209
PHP	0	37472	428.99	154
Python	0	12550	485.40	208
Ruby	1	4165	287.92	131
TypeScript	1	9824	332.81	161.5
Vim Script	317	10839	3553.74	3578

Table 4.8 highlights the variation in the size of code changes (measured in lines of code) required to fix vulnerabilities in different programming languages. For example, C projects have shown the widest range in fix sizes, with some fixes being minimal and others substantial, suggesting a high variability in the complexity of vulnerabilities in C. Meanwhile, languages like Ruby and TypeScript tend to have smaller and more consistent fix sizes, as indicated by their lower mean and median values. Notably, Vim Script stands out with a remarkably high mean and median, indicating that fixes in this language tend to be significantly larger than in other languages.

Table 4.9: Complexity of the changed files by programming languages.

Programming language	**Min**	**Max**	**Mean**	**Median**
C	0	10284	259.73	132
C++	0	3485	128.91	42
Go	0	13249	125.21	26
Java	0	1561	66.75	22
JavaScript	0	50897	932.15	39
PHP	0	18601	85.19	17
Python	0	3022	91.26	34
Ruby	0	606	41.96	18
TypeScript	0	1249	43.68	19
Vim Script	0	3098	693.33	460

Table 4.9 presents the complexity of vulnerability fixes in terms of cyclomatic complexity, a metric that evaluates the number of linearly independent paths through a program.

The data indicates that JavaScript and Vim Script have the highest potential for complex fixes, with JavaScript having an extraordinarily high maximum complexity (50,897) compared to other languages. This suggests that while JavaScript fixes may not always be extensive in terms of line count, they can be highly intricate.

Table 4.10: Token count by programming languages.

Programming language	**Min**	**Max**	**Mean**	**Median**
C	0	142811	7900.51	4250
C++	0	117238	5419.44	2284.5
Go	2	291613	4432.67	1309
Java	4	42889	2860.18	1122
JavaScript	0	2383237	37680.39	1481
PHP	0	435017	2938.16	1006.5
Python	0	103942	3271.09	1323.5
Ruby	2	27233	1693.07	699
TypeScript	7	47976	2183.29	1056
Vim Script	862	68652	14882.42	13082

Table 4.10 shows the verbosity or syntactic complexity of the fixes across languages. The collected data suggests that some vulnerability fixes in JavaScript can be highly verbose or involve significant syntactic changes. On the other hand, programming languages like Ruby and Python have considerably lower mean and median token counts, indicating less syntactic complexity in the fixes. Vim Script shows a high mean and median, aligning with its high NLOC and complexity values, suggesting more extensive and complex fixes in this language.

Chapter 5

Conclusion and Future Works

This work presented a comprehensive study of various aspects related to software vulnerabilities and the characteristics of their fix commits. A large dataset containing essential information about 11,558 vulnerabilities in 58 programming languages and their respective patch codes in 11,332 commits across 3,587 projects was developed. The analyses carried out on the constructed dataset showed that specific languages are more prone to certain types of vulnerabilities, such as PHP for web and network application vulnerabilities and C/C++ for memory and resource management vulnerabilities.

It was also found that there is significant variability in the frequency and types of vulnerabilities across different programming languages. Languages such as C and JavaScript showed a wide range in the size and complexity of fixes. In contrast, languages such as Ruby and TypeScript tended to have smaller fix sizes and a higher proportion of low-risk changes. This suggests that fixes in these languages involve less complex methods and are easier to maintain. However, JavaScript had the highest potential for complex fixes, with fixes showing extensive verbosity or significant syntactic changes.

Additionally, there are several possible applications for the data collected: the dataset can be used to apply more advanced statistical or machine learning techniques to discover deeper insights and potentially predictive models about software vulnerabilities and their fixes. Another possible application involves a deeper analysis of vulnerability patterns specific to languages such as C, JavaScript, and PHP, which could help create more targeted and effective analysis tools for each language.

Other applications include comparative studies to understand why certain languages tend to have lower-risk features while others have more complex, higher-risk features and how this correlates with language standards and its use in various projects. It is also possible to use the dataset to explore vulnerabilities presented in multiple languages, seeking to understand how vulnerabilities in one language can affect or be affected by codes developed in another language.

One of the limitations of this work is the choice of projects only available on GitHub to maximize and standardize metadata extraction. Future work could consider this and

expand the sources to collect projects hosted on the GitLab, BitBucket, or Sourceforge platforms and cover other bug and vulnerability tracking platforms such as Bugzilla, Mantis, and Redmine. Future work may also include an analysis through a manual review of the collected source code to provide a more solid basis for evaluating security threats and their detection and fixes processes.

Bibliography

[Alves et al., 2016a] Alves, H., Fonseca, B., and Antunes, N. (2016a). Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. IEEE.

[Alves et al., 2016b] Alves, H., Fonseca, B., and Antunes, N. (2016b). Software metrics and security vulnerabilities: Dataset and exploratory study. In *2016 12th European Dependable Computing Conference (EDCC)*. IEEE.

[Bhandari et al., 2021] Bhandari, G., Naseer, A., and Moonen, L. (2021). CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM.

[Cawthra et al., 2020] Cawthra, J., Ekstrom, M., Lusty, L., Sexton, J., and Sweetnam, J. (2020). *Data Integrity: Identifying and Protecting Assets Against Ransomware and Other Destructive Events*.

[Challande et al., 2022] Challande, A., David, R., and Renault, G. (2022). Building a commit-level dataset of real-world vulnerabilities. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*. ACM.

[Coelho et al., 2014] Coelho, F. E. S., Araújo, L. G. S. d., and Bezerra, E. K. (2014). *Gestão da Segurança da Informação: NBR 27001 e NBR 27002*. Rede Nacional de Ensino e Pesquisa – RNP/ESR, Rio de Janeiro.

[Dempsey et al., 2020] Dempsey, K., Takamura, E., Eavy, P., and Moore, G. (2020). Automation support for security control assessments:. Technical report.

[di Biase et al., 2019] di Biase, M., Rastogi, A., Bruntink, M., and van Deursen, A. (2019). The delta maintainability model: Measuring maintainability of fine-grained code changes. In *Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019*, Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019, pages 113–122, United States. IEEE. TechDebt 2019 - International Conference on Technical Debt, TechDebt 2019 ; Conference date: 26-05-2019 Through 27-05-2019.

[Erbel and Kopniak, 2018] Erbel, M. and Kopniak, P. (2018). Assessment of the web application security effectiveness against various methods of network attacks. *Journal of Computer Sciences Institute*, 9:340–344.

[Fan et al., 2020] Fan, J., Li, Y., Wang, S., and Nguyen, T. N. (2020). A c/c code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM.

[Joh and Malaiya, 2017] Joh, H. and Malaiya, Y. K. (2017). Periodicity in software vulnerability discovery, patching and exploitation. *International Journal of Information Security*, 16:673–690.

[Le et al., 2021] Le, T. H. M., Croft, R., Hin, D., and Babar, M. A. (2021). A large-scale study of security vulnerability support on developer q&a websites. In *Evaluation and assessment in software engineering*, pages 109–118.

[Lin et al., 2023] Lin, J., Zhang, H., Adams, B., and Hassan, A. E. (2023). Vulnerability management in linux distributions: An empirical study on debian and fedora. *Empirical Software Engineering*, 28(2):47.

[Luo et al., 2020] Luo, C., Bo, W., Kun, H., and Yuesheng, L. (2020). Study on software vulnerability characteristics and its identification method. *Mathematical Problems in Engineering*, 2020:1–6.

[McCabe, 1976] McCabe, T. J. (1976). A complexity measure. *IEEE Trans. Software Engineering*, 2(4):308–320.

[MITRE, 2023a] MITRE (2023a). Cve security vulnerability database.

[MITRE, 2023b] MITRE (2023b). Cwe top 25 most dangerous software weaknesses.

[Nikitopoulos et al., 2021] Nikitopoulos, G., Dritsa, K., Louridas, P., and Mitropoulos, D. (2021). Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21. ACM.

[Paulsen and Byers, 2019] Paulsen, C. and Byers, R. (2019). *Glossary of key information security terms*.

[Pianco et al., 2016] Pianco, M., Fonseca, B., and Antunes, N. (2016). Code change history and software vulnerabilities. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE.

[Shahzad et al., 2012] Shahzad, M., Shafiq, M. Z., and Liu, A. X. (2012). A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE.

[Sêmola, 2013] Sêmola, M. (2013). *Gestão da Segurança da Informação*. ISBN 978-85-352-1191-7.