FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# End-to-end Approach for Automated Vulnerability Identification and Patching

**Eduard Costel Pinconschi**

WORKING VERSION


Programa Doutoral em Engenharia Informática

Supervisor: Prof. <Name of the Supervisor>

June 15, 2025

# End-to-end Approach for Automated Vulnerability Identification and Patching

**Eduard Costel Pinconschi**

Programa Doutoral em Engenharia Informática

June 15, 2025

# Resumo

Este documento ilustra o formato a usar em dissertações na Feup. São dados exemplos de margens, cabeçalhos, títulos, paginação, estilos de índices, etc. São ainda dados exemplos de formatação de citações, figuras e tabelas, equações, referências cruzadas, lista de referências e índices. Este documento não pretende exemplificar conteúdos a usar. É usado o *Loren Ipsum* para preencher a dissertação. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam vitae quam sed mauris auctor porttitor. Mauris porta sem vitae arcu sagittis facilisis. Proin sodales risus sit amet arcu. Quisque eu pede eu elit pulvinar porttitor. Maecenas dignissim tincidunt dui. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec non augue sit amet nulla gravida rutrum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nunc at nunc. Etiam egestas. Donec malesuada pede eget nunc. Fusce porttitor felis eget mi mattis vestibulum. Pellentesque faucibus. Cras adipiscing dolor quis mi. Quisque sagittis, justo sed dapibus pharetra, lectus velit tincidunt eros, ac fermentum nulla velit vel sapien. Vestibulum sem mauris, hendrerit non, feugiat ac, varius ornare, lectus. Praesent urna tellus, euismod in, hendrerit sit amet, pretium vitae, nisi. Proin nisl sem, ultrices eget, faucibus a, feugiat non, purus. Etiam mi tortor, convallis quis, pharetra ut, consectetuer eu, orci. Vivamus aliquet. Aenean mollis fringilla erat. Vivamus mollis, purus at pellentesque faucibus, sapien lorem eleifend quam, mollis luctus mi purus in dui. Maecenas volutpat mauris eu lectus. Morbi vel risus et dolor bibendum malesuada. Donec feugiat tristique erat. Nam porta auctor mi. Nulla purus. Nam aliquam.

# Abstract

Here goes the abstract written in English. Should say the same.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Sed vehicula lorem commodo dui. Fusce mollis feugiat elit. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec eu quam. Aenean consectetuer odio quis nisi. Fusce molestie metus sed neque. Praesent nulla. Donec quis urna. Pellentesque hendrerit vulputate nunc. Donec id eros et leo ullamcorper placerat. Curabitur aliquam tellus et diam. Ut tortor. Morbi eget elit. Maecenas nec risus. Sed ultricies. Sed scelerisque libero faucibus sem. Nullam molestie leo quis tellus. Donec ipsum. Nulla lobortis purus pharetra turpis. Nulla laoreet, arcu nec hendrerit vulputate, tortor elit eleifend turpis, et aliquam leo metus in dolor. Praesent sed nulla. Mauris ac augue. Cras ac orci. Etiam sed urna eget nulla sodales venenatis. Donec faucibus ante eget dui. Nam magna. Suspendisse sollicitudin est et mi. Fusce sed ipsum vel velit imperdiet dictum. Sed nisi purus, dapibus ut, iaculis ac, placerat id, purus. Integer aliquet elementum libero. Phasellus facilisis leo eget elit. Nullam nisi magna, ornare at, aliquet et, porta id, odio. Sed volutpat tellus consectetuer ligula. Phasellus turpis augue, malesuada et, placerat fringilla, ornare nec, eros. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Vivamus ornare quam nec sem mattis vulputate. Nullam porta, diam nec porta mollis, orci leo condimentum sapien, quis venenatis mi dolor a metus. Nullam mollis. Aenean metus massa, pellentesque sit amet, sagittis eget, tincidunt in, arcu. Vestibulum porta laoreet tortor. Nullam mollis elit nec justo. In nulla ligula, pellentesque sit amet, consequat sed, faucibus id, velit. Fusce purus. Quisque sagittis urna at quam. Ut eu lacus. Maecenas tortor nibh, ultricies nec, vestibulum varius, egestas id, sapien. Donec hendrerit. Vivamus suscipit egestas nibh. In ornare leo ut massa. Donec nisi nisl, dignissim quis, faucibus a, bibendum ac, diam. Nam adipiscing hendrerit mi. Morbi ac nulla. Nullam id est ac nisi consectetuer commodo. Pellentesque aliquam massa sit amet tellus. Vivamus sodales aliquam leo.

# Acknowledgements

Aliquam id dui. Nulla facilisi. Nullam ligula nunc, viverra a, iaculis at, faucibus quis, sapien. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Curabitur magna ligula, ornare luctus, aliquam non, aliquet at, tortor. Donec iaculis nulla sed eros. Sed felis. Nam lobortis libero. Pellentesque odio. Suspendisse potenti. Morbi imperdiet rhoncus magna. Morbi vestibulum interdum turpis. Pellentesque varius. Morbi nulla urna, euismod in, molestie ac, placerat in, orci.

Ut convallis. Suspendisse luctus pharetra sem. Sed sit amet mi in diam luctus suscipit. Nulla facilisi. Integer commodo, turpis et semper auctor, nisl ligula vestibulum erat, sed tempor lacus nibh at turpis. Quisque vestibulum pulvinar justo. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nam sed tellus vel tortor hendrerit pulvinar.

Fusce gravida placerat sem. Aenean ipsum diam, pharetra vitae, ornare et, semper sit amet, nibh. Nam id tellus. Etiam ultrices.

<O Nome do Autor>

*"Our greatest glory is not in never falling, but in rising every time we fall"*

Confucius

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **APR** | Automatic Program Repair |
| **ASR** | Automatic Software Repair |
| **API** | Application Programming Interface |
| **CIA** | Confidentiality, Integrity, and Availability |
| **CGC** | Cyber Grand Challenge |
| **CPE** | Common Platform Enumeration |
| **CVE** | Common Vulnerabilities and Exposures |
| **CWE** | Common Weakness Enumeration |
| **DARPA** | Defense Advanced Research Projects Agency |
| **FL** | Fault Localization |
| **ML** | Machine Learning |
| **NMT** | Neural Machine Translation |
| **NVD** | National Vulnerability Database |
| **OS** | Operating System |
| **SDLC** | Software Development Life Cycle |
| **POV** | Proof of Vulnerability |

# Chapter 1

# Introduction

## 1.1 Research Directions

In this thesis, we explore three main research directions as follows:

### Automated Analysis and Mining of Vulnerabilities

Contemporary vulnerability research remains siloed: fragmented taxonomies address only isolated aspects of faults (e.g., causes or resource constraints) without unifying code-level patterns and system-wide contexts, historical vulnerability repositories (e.g., NVD, OSV, GitHub Advisory) offer abundant data but lack automated tools for extracting and labeling actionable code vulnerabilities, and existing benchmarks (e.g., Defects4J, Vul4J) reproduce faults yet fail to mirror actual distributions across languages, technologies, and application domains. Our research direction seeks to bridge these gaps by developing an automated, data-driven framework that integrates multi-level profiling of vulnerabilities, leverages scalable mining of historical feeds to identify and annotate code-level weakness patterns, and synthesizes benchmarks whose composition reflects the true landscape of software faults. By aligning classification, automation, and benchmark construction, this approach aims to create a coherent ecosystem for understanding, evaluating, and ultimately improving software security.

### Machine learning-based SAST

Machine learning-based Static Application Security Testing (SAST) represents a paradigm shift from traditional rule-based vulnerability detection to data-driven approaches that can automatically learn patterns and characteristics of security vulnerabilities from large codebases. Unlike conventional SAST tools that rely on predefined rules and signatures to identify known vulnerability patterns, ML-based Software Vulnerability Prediction (SVP) systems leverage deep learning models, neural networks, and ensemble techniques to analyze source code and predict potential security weaknesses. These approaches promise to address fundamental limitations of traditional

static analysis, including high false positive rates, limited adaptability to new vulnerability types, and the need for extensive expert knowledge to configure and maintain detection rules.

The motivation for exploring complementary trade-offs between SAST and Software Vulnerability Detection (SVD) approaches stems from their inherent strengths and limitations. Traditional SAST tools offer precision and explainability but struggle with novel vulnerability patterns, while ML-based SVD systems excel at pattern recognition but often lack interpretability and suffer from dataset biases. By investigating how these approaches can complement each other, this research aims to develop hybrid systems that combine rule-based precision with ML-based adaptability. Such integration could significantly reduce false positives while improving detection capabilities for emerging vulnerability types, ultimately creating more robust and practical security tools for real-world software development environments.

### Benchmarking SAST-Based APR on Software Vulnerabilities

The Open Source Software (OSS) ecosystem is a vibrant and collaborative environment comprising independent developers, enterprises, and academic contributors who work together to build software used in nearly every domain. Despite advances in development practices, security vulnerabilities remain a significant challenge, often discovered only after exploitation. Automated Program Repair (APR) for software vulnerabilities represents a promising approach to address this challenge, offering the potential to automatically generate patches for identified security weaknesses.

This research direction is motivated by the need to systematically evaluate and improve APR tools specifically for vulnerability remediation. By focusing on memory issues, data validation failures, and faulty computations in C, C++, and Java programs, this work aims to establish comprehensive benchmarks that reflect real-world vulnerability scenarios. The integration of Software Vulnerability Detection (SVD) techniques with APR systems has the potential to create more effective end-to-end solutions that not only identify vulnerabilities but also automatically generate appropriate fixes. Such integration could significantly reduce the security burden on developers and maintainers, enabling more proactive and systematic approaches to software security across the OSS ecosystem.

In this thesis, we also aim to answer the following high-level research questions:

**RQ.1** *How can automated vulnerability analysis shape models of software security and real-world benchmarks?*

The software security landscape is vast and complex, demanding a unified framework to profile faults across different layers—from languages to system components—to make sense of their diversity. However, profiling alone is not sufficient: to translate insights into practical benchmarks, we must isolate actionable vulnerabilities with concrete code artifacts that can be reproduced and tested. Achieving this at scale requires automated methods to mine historical vulnerability data, analyze source repositories, and label code patterns that signify real weaknesses. By combining comprehensive profiling with automated identification and annotation, we can construct benchmarks whose composition mirrors true fault distributions and supports reliable tool evaluation and

comparative studies.

**RQ.2** *What are the limitations of APR in repairing real-world software vulnerabilities, and can integrating SVD techniques like SAST help overcome them, particularly in coverage and fault localization?*

Despite significant advances in both Automated Program Repair (APR) and Software Vulnerability Detection (SVD) techniques such as Static Application Security Testing (SAST), a clear gap remains in understanding and improving their effectiveness on real-world software vulnerabilities. Existing studies often evaluate APR on general defects rather than security-specific issues, while SAST evaluations are frequently limited to synthetic benchmarks like SARD, raising concerns about generalizability. Moreover, APR suffers from persistent challenges like incomplete fault localization and overfitting to test cases, which undermine its reliability and scalability. SVD techniques, by contrast, can identify specific vulnerability patterns and provide semantic insights without execution, making them promising candidates to address APR's bottlenecks. However, few empirical studies explore the integration of these techniques to enhance vulnerability coverage, localization accuracy, and patch quality. This research is motivated by the need to benchmark APR tools specifically on real-world vulnerabilities and to investigate whether incorporating SVD methods can mitigate known limitations, thereby advancing the state of vulnerability repair in practice.

## 1.2 Dissertation Structure

In addition to the introduction, this dissertation contains 8 more chapters. Chapter 2 reviews the relevant background and related work in vulnerability detection and program repair. Chapter 3 presents the profiling of security faults and discusses patterns observed across real-world software. Chapter 4 describes the construction of the benchmark dataset used in subsequent experiments. Chapter 5 provides a comparative analysis of existing APR and SVD tools in terms of their vulnerability coverage and performance. Chapter 6 introduces and evaluates the hybrid repair approach combining APR with static vulnerability detection. Chapter 7 explores the integration of ML-based detection with SAST tools in a collaborative setting. Chapter 8 discusses the key findings, implications, and limitations of the work. Chapter 9 concludes the thesis and outlines directions for future research.

# Chapter 2

# Profiling Software Security Faults

This chapter presents the framework, methodology, and preliminary findings of our study on profiling software security faults across multiple dimensions (software type, technology, causes, and programming language). **Eduard:** to be continued END

## 2.1 Introduction

Understanding software vulnerabilities is central to improving system resilience and guiding secure software engineering practices. Security faults represent fundamental weaknesses in the construction of software systems that adversaries can exploit to compromise functionality, confidentiality, integrity, or availability. Despite decades of research, the prevalence and evolution of such faults remain only partially understood.

One of the current challenges lies in the limited understanding of the prevalence and structure of security faults across software systems. Numerous studies have attempted to organize vulnerability data and detect recurring patterns, typically driven by specific objectives: identifying root causes, building automated detection tools, or framing vulnerabilities within particular operational contexts. However, such efforts often suffer from scope limitations or overly narrow classification schemes.

A more comprehensive approach is needed—one that captures the complexity of security faults by profiling them through multiple dimensions. We posit that a granular, multi-dimensional profiling of vulnerabilities—considering attributes such as the root cause, programming language, affected components, and software type—can offer a deeper insight into the software security landscape. Classifying large numbers of vulnerability instances across distinct levels of software granularity enables a systematic decomposition of their complexity, helping bridge the gap between isolated technical analyses and broader software security assessments.

Preliminary investigations highlight this gap. Existing classification schemes are predominantly attack- or operating-system-oriented, with only a minority addressing the software level directly. For instance, among 25 classification methodologies surveyed, only three focus on software characteristics. Prior efforts, such as those by Landwehr et al., Garg et al., and Ezenwoye et

al., represent valuable foundations. Yet, none provide a unified profiling that scales from the programming language level up to broader software categories. As vulnerabilities increasingly span multiple layers of abstraction and technology, such profiling becomes vital to threat modeling, secure design, and targeted mitigation strategies.

### 2.1.1 Research Question and Objectives

This chapter addresses the following overarching research question:

> *How do software security faults profile regarding software type, technology, causes, and programming language?*

To answer this, we explore several subquestions:

- How do vulnerability patterns differ across software types (e.g., web applications, utilities, operating systems)?
- How do programming languages correlate with the occurrence of security faults?
- Which technologies (e.g., frameworks, protocols, libraries) are most commonly affected by specific fault classes?
- What are the leading causes (e.g., improper input validation, memory management issues) associated with each vulnerability category?

The primary objective is to profile security faults at four distinct levels of software granularity by identifying empirical patterns in publicly available vulnerability repositories. This approach aims to reveal associations between intrinsic software characteristics and the types of security faults observed, offering actionable insights into their root causes and distribution.

### 2.1.2 Scope and Contributions

This work focuses on vulnerabilities disclosed over the past 25 years, as cataloged by the National Vulnerability Database (NVD). We restrict our analysis to application-level vulnerabilities, filtering out Common Vulnerabilities and Exposures (CVE)s marked as Rejected, Deferred, Disputed, or Unsupported, and considering only those with a valid CPE designation and an associated primary weakness.

The key contributions of this chapter include:

- A unified and holistic taxonomy of software vulnerabilities that intersects and extends existing classification frameworks.
- Empirical patterns of fault occurrence across software types, technologies, causes, and programming languages, illuminating multi-level software susceptibility.

- A reproducible data collection and analysis pipeline that supports future vulnerability research and classification efforts.

By providing a systematic, multi-dimensional profile of security faults, this work aims to enhance our understanding of software vulnerability trends and support more effective strategies in secure software design, maintenance, and threat modeling.

## 2.2 Background and Definitions

### 2.2.1 Software Security

**Eduard:** introduce Software Security and then elaborate on the properties END

Jeff Hughes and George Cybenko [1] identify three elements for computing system threats to existing:

- inherent system susceptibility;
- the access of the threat to the susceptibility;
- the capability of threat to exploit the susceptibility.

Securing the system in the first place can reduce the susceptibility to the threat. The concept of security in computing systems encompasses three attributes: *availability*, *integrity*, and *confidentiality* [2]. The *IEEE Standard Glossary of Software Engineering Terminology* [3] defines the first two attributes as follows:

**Definition 2.2.1** (Availability)**.** *"The degree to which a system or component is operational and accessible when required for use."*

**Definition 2.2.2** (Integrity)**.** *"The degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data."*

The National Institute of Standards and Technology (NIST) defines the last attribute as following [4]:

**Definition 2.2.3** (Confidentiality)**.** *"Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information."*

Thus, accommodating a system with Confidentiality, Integrity, and Availability (CIA) properties dictates its level of security. Jeff Hughes and George Cybenko [1] claim that absolute CIA is unachievable as systems inevitably will have design trade-offs resulting in inherent weaknesses that can manifest, for instance, faults. In software, the concept of security is about designing fault-free and fault-tolerant software so it can continue functioning correctly even under attack [5]. Software security also has an educational component requiring developers, architects, and users to follow best practices in software engineering and its usage.

### 2.2.2 Software Security Faults

A *fault* is the root cause of an *error* that may lead to *failure*(s) [6]. A fault can be internal or external to a system. The former concerns its internal state, while the latter is its external state, perceivable at the interface of a system. The *IEEE Standard Glossary of Software Engineering Terminology* [3] includes the following definitions for the previous terms:

**Definition 2.2.4** (Fault)**.** *"An incorrect step, process, or data definition in a computer program.".*

**Definition 2.2.5** (Error)**.** *"A human action that produces an incorrect result." (this definition is assigned to the word "mistake").*

**Definition 2.2.6** (Failure)**.** *"The inability of a system or component to perform its required functions within specified performance requirements").*

Figure 2.1 depicts the manifestation mechanism of fault, error, and failure. It starts when a computation activates the fault, and the system enters an incorrect state. In that state, the error occurs as the system deviates from its intended functionality. An error propagates until it passes through the system-user interface, then becomes a *failure*.

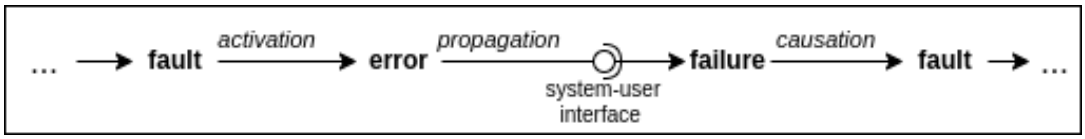


Figure 2.1: The manifestation chain of a fault (*cf.* [2]).

Faults cause expected scenarios not to run and turn into security faults when they compromise a system's security attributes, *i.e.*, CIA. The examples in Figure 2.2 depicts the difference. The fault in listing 2.1 does not allow the *pixels* variable to increment. On the other hand, the *security fault* in listing 2.2 results from wrong bounds-checking the *for* cycle and subjects the system to a memory error. The error can be exploited by loading shell code into the allocated buffer memory, exposing the system to potential malicious use cases. A security fault that is accessible and exploitable is a *vulnerability*. We will elaborate on vulnerabilities in section **??**.

accidental semicolon

```
1 for (i=0; i<numrows; i++)
2     for (j=0; j<numcols; j++);
3         pixels++;
```

Listing 2.1: Fault example (*cf.* [7]).

off-by-one error

```
1 void vuln(){
2         int i;
3         int buf[128];
4
5         for (i=0; i <= 128; i++)
6             cin >> buf[i];
7 }
```

Listing 2.2: Security fault example (*cf.* [8]).

Figure 2.2: Example of a fault (left) and a security fault (right).

To enforce security policies, Landwehr *et al.* [6] characterize security faults by three dimensions: *genesis*, *time of introduction*, and *location*. The former answers how a security fault entered the system and whether it can be *intentional* or *inadvertently*. The time of introduction of a security fault aligns with the Software Development Life Cycle (SDLC). It is determined by one of the three different phases in the life cycle of a system: *development*, *maintenance*, or *operational*. For Avizienis *et al.* [2], the time of introduction resumes with the *development* and *operational* phases, as the latter covers maintenance. During development, all activities up to the release of the initial version of the system can originate security faults, for instance, by including additional mechanisms to the specification for testing the system. During the maintenance phase, activities leading to security faults are often attributable to malicious intrusion or incompetence of the programmer in understanding the system. During the operational phase, unauthorized or unsafe modifications to the system can introduce security faults — *e.g.*, installing virus programs. Finally, a security fault starts manifesting in a location in the system, which can be in the *software* or *hardware*.

Our focus regarding genesis is on accidental security faults as in the latter, the detection approaches are much less likely to help, and measures concern the trustworthiness of the programmers. Regarding the time of introduction and location, our concern is software security faults introduced during the development phase. These can originate in *requirements and specifications*, *source code*, or *object code*. Software requirements and specifications only describe the design of a particular system and its functionality, source code is the actual implementation, and object code only represents the machine-readable form of the previous. We consider source code as we focus on avoiding faults in the software during its implementation to handle threats proactively.

A security fault in software can occur in one of the following software layers: *operating system programs*, *support software*, or *application software*. Similarly, Khwaja *et al.* [9] identify, based on mitigation techniques, the reach of security faults in software but only at two levels: *operating systems* and *software applications*. Security faults in the former can compromise system functions, such as process and memory management, and cause the most significant injury. Support and application software are programs outside the Operating System (OS) boundary — *e.g.*, editors and libraries. Support software has granted privileges, and a security fault in them can allow privilege escalation to gain further unauthorized access to the system. Application software has no special system privileges, but a security fault in them can still cause considerable damage within the storage of the victim.

### 2.2.3 Classes of Software Security Faults

**Eduard:** TODO END

## 2.3 Related Work

Several studies suggest vulnerability taxonomies oriented toward particular scopes, such as prevalent vulnerability types in specific operating systems [10]. Joshi *et al.* [11] summarize and compare

the characteristics of 25 taxonomies of attacks and vulnerabilities in computer and network systems. Most taxonomies are attack or OS oriented, and only three [12]–[14] are software oriented. Their analysis indicates that none of the taxonomies satisfy all the necessary principles about classifications. In addition, the existing taxonomies are outdated and limited in use. A standard vulnerability classification method or scheme can increase the security assessment of software systems by providing a common language and a good overview of the field of study. In this section, we offer a summary of the different taxonomies.

Table 2.1: Summary Table of Reviewed Taxonomies

| Year | Taxonomy | Scheme | Attributes | Objective |
|---|---|---|---|---|
| 1994 | Landwehr *et al.* [6] | Multi-dimensional | genesis, time, location | establish taxonomy of security faults |
| 2002 | Frank Piessens [13] | Hierarchical | genesis, time | understand mistakes of software developers |
| 2005 | Weber *et al.* [14] | Hierarchical | genesis | aid designers of code analysis tools |
| 2007 | Bazaz & Arthur [12] | Hierarchical | resources, properties | strategies to evaluate software security |
| 2017 | Goseva & Tyo [15] | Multi-Dimensional | genesis, severity, location, time | build evidence-based vulnerability knowledge |
| 2019 | Garg *et al.* [16] | Multi-dimensional | genesis, severity, platform, technique | predictive modeling & preemptive mitigation |
| 2020 | Ezenwoye *et al.* [17] | Single-dimensional | software type | reveal prevalence & persistence patterns |
| ... | ... | ... | ... | ... |

Frank Piessens [13] proposes a structured taxonomy focusing on the causes of software vulnerabilities to foster the identification of vulnerabilities during software review. Weber *et al.* [14] propose a taxonomy of software security faults oriented toward designing code analysis tools. Their taxonomy leverages Landwehr's work [6] and descriptions of vulnerabilities and threats. They re-design the prior classification of security faults by genesis and remove faults not relevant to code analysis tools, such as configuration errors. Anil Bazaz and James Arthur [12] present a taxonomy of vulnerabilities for assessing software security with verification and validation strategies. Their scheme uses computer resources as the top categories of the taxonomy, covering memory, I/O, and cryptographic resources. The bottom level of the taxonomy conveys violable constraints and assumptions, which allows checking if a software application permits the violation of such constraints and assumptions.

Garg *et al.* [16] classify software vulnerabilities based on software systems, severity level, techniques, and causes. Their classification scheme intends to give a comprehensive view of vulnerabilities in various domains, as vulnerabilities can happen for several reasons in a system. Goseva-Popstojanova and Tyo [15] introduce an empirically driven classification framework that combines CWE-888 Software Fault Pattern (SFP) View with lifecycle phase analysis to create actionable vulnerability profiles. Ezenwoye *et al.* [17] classify 51,110 vulnerability entries from

the NVD database into seven software types (OS, browser, middleware, utility, web application, framework, and server). Their analysis demonstrates the pattern of prevalence of software faults by software type. Subsequently, Ezenwoye *et al.* [18] review ten years of vulnerability data in the NVD database to profile the most common web application faults according to three attributes: attack method, attack vectors, and technology. The latter attribute captures the susceptibility of technologies to faults, including programming languages, frameworks, communication protocols, and data formats. **Eduard:** there are more recent works that need to be added, e.g., Software Vulnerability Analysis Across Programming Language and Program Representation Landscapes: A Survey END

### 2.3.1 Identified Gaps

**Eduard:** needs to be validated/aligned with the related work above and the gap analysis in the proposal END Despite these efforts, several important gaps remain:

- **OS- and attack-centric focus:** Existing taxonomies predominantly emphasize operating systems or particular attack vectors (for example, network attacks or web application exploits) [11]. There is little focus on vulnerabilities specific to application frameworks or general-purpose software contexts.
- **Scarcity of software-oriented taxonomies:** Very few taxonomies (only about 3 of the 25 reviewed by Joshi *et al.* [11]) explicitly classify vulnerabilities by the characteristics of the underlying software system. Most classification schemes do not consider factors such as software architecture, implementation language, or development environment, leaving a gap in understanding vulnerabilities at the application level.
- **Lack of multi-dimensional profiling:** Current classification schemes are largely unidimensional. No existing taxonomy jointly profiles vulnerabilities by intersecting factors (for example, programming language *and* fault type, or software layer and vulnerability cause). This lack of intersectional profiling means we cannot directly relate specific software attributes (such as language or framework) to the kinds of security faults they most frequently exhibit.
- **Absence of unified empirical methods:** There is no single unified, data-driven approach that ties vulnerability records systematically to software characteristics (such as code metrics, library usage, or development practices). Existing approaches either analyze raw CVE data by one attribute (like severity) or propose ad-hoc taxonomies, but none provide an empirical profiling of vulnerabilities linked to software attributes.

### 2.3.2 Summary of Research Gap

In summary, no single comprehensive taxonomy currently exists that profiles software vulnerabilities across multiple levels of granularity and multiple classification dimensions simultaneously.

Existing schemes tend to cover only specific domains or one dimension at a time, leaving gaps in our understanding of how vulnerabilities correlate with software characteristics. This gap implies that threat modeling and design-time security decisions often lack systematic guidance on which weaknesses are likely in which kinds of software. A more holistic, multi-dimensional vulnerability profiling approach is needed to support effective risk assessment and mitigation. In particular, a unified taxonomy linking vulnerability types to software context and implementation details would enable practitioners to anticipate likely security faults and tailor their prevention strategies across the software development lifecycle [11].

## 2.4 Methodology

The methodology is structured into three sequential phases, each addressing a distinct part of our multi-dimensional analysis. In Phase I we develop a holistic taxonomy-based representation of security faults. In Phase II we construct an automated data-extraction pipeline to collect and aggregate vulnerability records. In Phase III we perform exploratory analysis on the dataset and derive a hierarchical classification scheme based on observed patterns. Each phase builds on the previous, ensuring a coherent flow from abstract taxonomy to concrete classification.

### 2.4.1 Holistic Representation of Security Faults

**Aim:** Synthesize existing vulnerability taxonomies into a unified attribute schema. This phase consolidates diverse security-fault taxonomies (e.g. language-specific faults, technology categories, architectural layers, and root-cause taxonomies) to create a comprehensive representation of vulnerability attributes.

**Steps:**

- **Survey taxonomies.** We collect relevant taxonomies from literature and standards. For example, language- and platform-based taxonomies (e.g. C/C++ errors vs. Web app issues), technology-focused taxonomies (e.g. web, database, OS level), software-layer taxonomies (e.g. presentation vs. business logic), and root-cause taxonomies (e.g. input validation, memory corruption).
- **Identify attributes.** From each taxonomy we extract key attributes (e.g., "Buffer Overflow", "SQL Injection", "Cross-Site Scripting", "Race Condition", etc.) and note which taxonomies include each attribute.
- **Create Unified Attribute Matrix.** We compile these attributes into a matrix with rows as example attributes and columns indicating taxonomy source. Table 2.2 (excerpt below) illustrates how attributes map across taxonomies. This "Unified Attribute Matrix" highlights overlaps and gaps, guiding later analysis.

Table 2.2: Unified Attribute Matrix

| Attribute | Lang-based [19] | Tech-based [9] | Layer [20] | Root Cause [21] |
|---|---|---|---|---|
| Buffer Overflow | ✓ | ✓ | | ✓ |
| SQL Injection | | ✓ | ✓ | |
| Cross-site Scripting | | ✓ | ✓ | |
| Privilege Escalation | ✓ | | | ✓ |
| Invalid Input | | | | ✓ |

This phase yields a consolidated schema of security fault attributes that will guide data collection and later classification. In particular, the unified list of attributes becomes the basis for labeling and grouping vulnerabilities in subsequent phases.

### 2.4.2 Collecting and Analyzing Data for Vulnerability Profiling

Figure **??** gives an overview of our multi-stage data processing pipeline for constructing a large-scale, multi-dimensional vulnerability dataset. This pipeline guides the systematic collection, enrichment, and consolidation of vulnerability data to support empirical profiling of security faults across software ecosystems. Our methodology integrates heterogeneous sources and leverages custom-developed tools to ensure consistency, traceability, and analytical utility of the resulting dataset.

The pipeline begins with **selecting application-level CVEs** enriched with code-relevant weaknesses. Drawing from over two decades of records in the National Vulnerability Database (NVD), we apply a reproducible filtering process that isolates vulnerabilities affecting application-layer software. This filtering is implemented using a suite of internally developed Python packages that enforce strict quality and relevance constraints, enabling a focused dataset suitable for downstream analysis. That includes `nvdutils`[1], `cpelib`[2], and `pydantic-cwe`[3].

Next, we perform **software type inference** to categorize affected products according to their functional roles—such as libraries, servers, or web applications. This classification enables more granular comparisons of vulnerability patterns across different categories of software. By analyzing metadata from the CPE dictionary and reconciling with curated external datasets, we construct a typology of software artifacts that improves the contextual interpretability of observed vulnerabilities.

To enhance this profiling further, we develop a methodology for **mapping software products to programming languages**. This stage involves inferring implementation languages through a combination of structured data extraction, curated mappings, and heuristics. Associating each

[1] `nvdutils`: A Python package for parsing, representing, filtering, and analyzing NVD data.

[2] `cpelib`: A Python package for parsing, representing, and filtering the NVD Common Platform Enumeration (CPE) Dictionary.

[3] `pydantic-cwe`: A Python package for modeling Common Weakness Enumeration (CWE) entries as Pydantic objects.

product with a language (or set of languages) allows us to investigate correlations between implementation choices and vulnerability types, addressing critical questions about the security implications of language ecosystems.

In the final stage, we perform **dataset consolidation**, merging enriched vulnerability records into a unified dataset. Each CVE is linked to a representative software product and annotated with associated metadata, including inferred language(s), software type, and CWE classification. This dataset—covering over 60,000 CVEs—enables visualization and analysis of security trends, such as frequent CWE-language pairings or high-risk software categories. A final co-occurrence analysis, visualized via a Sankey diagram, reveals dominant structural patterns in vulnerability distributions, laying the foundation for data-driven insights into the socio-technical determinants of software security.

#### 2.4.2.1 CVE and CWE Selection Criteria

We restricted the analysis exclusively to application-level vulnerabilities. This was operationalized using the configuration layer to include only products categorized under the "Application" part of the CPE taxonomy. This decision aligns with the research focus on software-level flaws rather than infrastructure or hardware vulnerabilities. To exclude incomplete, misleading, or invalid entries, the CVE filtering layer was configured to accept only entries marked as valid. Vulnerabilities flagged as Rejected, Deferred, Disputed, or Unsupported were automatically excluded. This ensures that the selected CVEs reflect accepted and actionable disclosures.

The dataset was further refined by retaining only CVEs associated with a single, deterministically selected weakness, thereby enhancing analytical precision and minimizing classification ambiguity. For CVEs mapped to multiple CWEs, a structured resolution strategy was applied based on guidance from MITRE's CWE usage recommendations. Each weakness was assigned a score according to its abstraction level, with more specific weaknesses prioritized, ordered from most to least specific as Variant > Base > Class > Compound > Chain. Additionally, weaknesses designated as "Primary" in the NVD schema were assigned an extra weight to further promote their selection. The CWE with the highest combined score was retained per CVE, ensuring consistency and mitigating noise from auxiliary or less informative mappings. To preserve semantic integrity and analytical relevance, weaknesses marked as deprecated or annotated with mapping notes falling outside the "Allowed" or "Allowed-with-Review" categories were excluded from consideration. This methodology reflects best practices in root-cause mapping and aligns with the CWE program's emphasis on using precise, non-discouraged mappings to represent the dominant weakness per vulnerability instance.

#### 2.4.2.2 Identification of Code-Relevant Weaknesses

To maintain thematic consistency with software implementation-level issues, we preselected a curated subset of CWEs that exhibit strong evidence of code-relevance. This subset was identified

by examining the nature of each weakness's lifecycle placement, detection modalities, presence of example code, and alignment with known software attack patterns:

- **Lifecycle Placement**: CWEs with introduction or mitigation phases classified as "Implementation" were prioritized, acknowledging that the origin or resolution of such weaknesses typically occurs within the software development lifecycle.
- **Detection Modalities**: CWEs amenable to automated or semi-automated static or dynamic analysis were included, consistent with scalable security tooling strategies.
- **Code Examples**: Weaknesses accompanied by reference implementations or exploit examples were retained as they provide tangible manifestations of the underlying flaw.
- **Attack Pattern Alignmen**t: CWEs linked to software attack pattern IDs from the CAPEC database were included to reinforce the practical exploitability of the weakness.

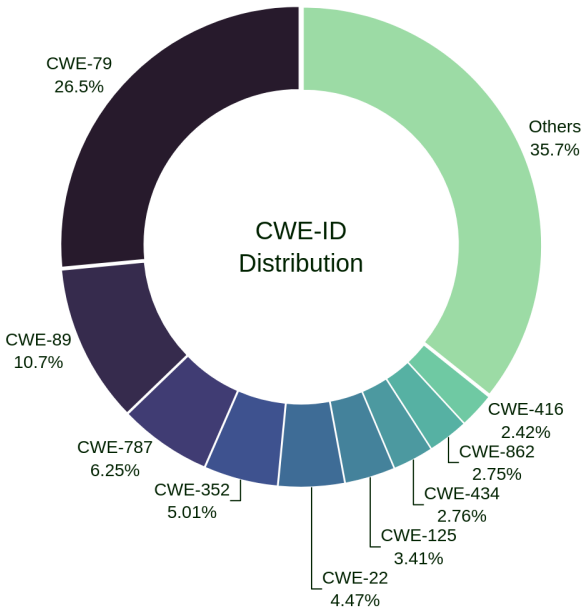


Figure 2.3: Distribution of CWE-IDs in the Selected CVE Set

The final script, executed over the full JSON feed of the NVD dataset, yielded 78,620 CVEs conforming to the above criteria. Each entry was recorded alongside its dominant CWE identifier. The top 25 CWEs accounted for over 83% of the total dataset, underscoring the concentration of application vulnerabilities around a core set of recurring weakness types. As shown in Figure 2.3, this concentration is particularly evident in the dominance of web application vulnerabilities, with CWE-79 (Cross-site Scripting) representing 26.5% of all entries, followed by CWE-89 (SQL Injection) at 10.7%. Memory safety issues also feature prominently, with CWE-787 (Out-of-bounds Write) accounting for 6.25% of vulnerabilities. Other significant categories include CWE-352 (Cross-Site Request Forgery) at 5.01% and CWE-22 (Path Traversal) at 4.47%,

while the remaining CWE types collectively comprise 35.7% of the dataset. This distribution enhances the tractability of downstream analyses and enables longitudinal and categorical insights into software vulnerability trends.

#### 2.4.2.3 Software Type Inference Based on CPE Metadata

The first stage involves labeling software entries in the CPE dictionary based on three complementary heuristics: (1) the target platform the software is associated with, (2) keywords in the product name, and (3) domain names in associated URLs.

- **Target Platform Mapping**: Certain CPE entries specify a `target_sw` field, which indicates the environment or platform that the software targets. We maintain a mapping from known platforms (e.g., wordpress, drupal, jira) to high-level software types such as extension, library, or framework. This mapping assumes that targeting a specific ecosystem typically implies a consistent functional role (e.g., WordPress-targeting software is likely a plugin or extension).
- **Product Name Keywords**: For entries lacking an informative target platform, we analyze the product name using a curated list of keywords associated with software types. Terms like plugin, sdk, module, and api are indicative of roles such as extension or library. This lexical heuristic captures semantic cues commonly embedded in naming conventions.
- **Reference Domain Analysis**: When a product includes external references (URLs), we extract domain names and match them against a precompiled mapping of known software directories and marketplaces (e.g., marketplace.visualstudio.com, patchstack.com). These sources are generally tied to specific types of software (e.g., extensions), and thus offer additional contextual signals.

Each of these signals contributes a possible label. If more than one label is inferred for a product, a resolution process is applied to select the most appropriate one.

#### 2.4.2.4 Label Reconciliation and Conflict Resolution

To strengthen the reliability of our labels, we align them with a manually labeled dataset from prior research by Ezenwoye *et al.* [17]. This dataset includes vendor and product pairs annotated with software types. After cleaning and normalizing the dataset (e.g., removing the category `operating system` and standardizing labels such as converting web application to `web_app`), we merge it with our inferred CPE dataset using vendor and product identifiers.

Merged entries may contain divergent labels from the CPE-based inference and the research dataset. In such cases, we apply a deterministic rule-based function to select one label based on the observed combination. These rules are designed to maintain consistency and prefer more specific labels when appropriate. For instance, if the research dataset labels a product as a framework, but our inference suggests library or extension, we retain the more granular label. In cases where the

CPE-inferred label is based solely on a reference URL (indicated by a `_ref` suffix), we prioritize that label if no better alternative exists.

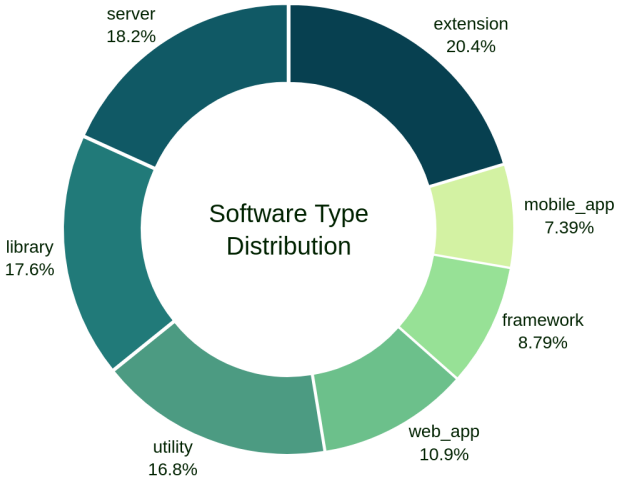


Figure 2.4: Distribution of Software Type in the Labelled Set of Products

Applying this classification methodology, we successfully labeled 34,575 unique software products with a corresponding software type. Figure 2.4 depicts the resulting distribution of functional roles: extension (20.35%), server (18.22%), library (17.57%), utility (16.82%), web_app (10.86%), framework (8.79%), and mobile_app (7.39%). This distribution highlights the prevalence of modular and infrastructure-supporting components such as extensions and libraries, as well as the substantial representation of utility and server software. These labeled categories form the foundation for our subsequent analyses of vulnerability types, enabling us to associate specific fault patterns with software of varying functional scopes and operational contexts.

#### 2.4.2.5 Mapping Software Products to Programming Languages

We begin by leveraging the publicly available SQLite database from the `purl2cpe` SCANOSS project [4], which maps package URLs (PURLs) to the standardized CPE identifiers. This dataset serves as a bridge between open-source software packages and the structured taxonomy used in vulnerability repositories. For each CPE entry, we extract associated PURLs and use predefined heuristics to infer programming languages. Specifically, we apply a static mapping from PURL types to canonical languages (e.g., pypi → Python, maven → Java). This heuristic captures common package ecosystems and provides a baseline for language classification.

However, not all packages yield direct mappings, particularly when repositories span multiple technologies or lack explicit package type indicators. To extend coverage, we incorporate

[4] `purl2cpe` - (https://github.com/scanoss/purl2cpe)

language metadata from GitHub repositories linked to the PURLs. This step employs a custom Python package developed for the study, coined `gitlib` [5], which interfaces with the GitHub API to identify the dominant programming language of a repository. The process includes parsing repository links, aggregating language-specific byte counts, and applying prioritization heuristics on a classification scheme during language inference when a dominant language is ambiguous. A final language label is assigned for each product, incorporating both direct mappings and inferred metadata.

#### 2.4.2.6 Programming Language Classification for Software Product Mapping

To accurately associate software products with their underlying programming languages, we classify programming languages into two main categories: **primary** and **secondary** languages. This distinction plays a crucial role in determining the most representative language for a given software product.

**Primary languages** encompass general-purpose programming languages capable of supporting the development of full-featured software systems. These languages are Turing-complete and are widely used across various domains, including web development, systems programming, and application development. When identifying a product's language, this category is prioritized due to its broad applicability and higher likelihood of representing the product's core logic. Examples of primary languages include:

- **Popular web and backend languages**: PHP, Python, Java, JavaScript;
- **Systems and application programming**: C, C++, Rust, Go, C#;
- **Scripting and object-oriented languages**: Ruby, Kotlin, Swift, TypeScript;
- **Functional and academic languages**: Haskell, Scala, Clojure, Erlang;
- **Legacy and enterprise languages**: Visual Basic, COBOL, Fortran, Ada;

**Secondary languages** are also executable but are tailored to specific domains or computational tasks. While they may appear frequently in repositories, their presence is often supplementary to the main application logic. As such, they are considered only when no suitable primary language can be confirmed. Examples of secondary languages include:

- **Database and query languages**: SQL, TSQL, PLpgSQL;
- **Statistical and scientific computing**: R, MATLAB, Mathematica;
- **System scripting and automation**: Shell, PowerShell;
- **Specialized domains**: Solidity (blockchain), TeX (typesetting);

[5] `gitlib` - https://github.com/epicosy/gitlib

#### 2.4.2.7 Product Language Prioritization Heuristics

When analyzing GitHub repositories associated with software products, the classification introduced above is used to guide language selection, particularly in cases where a software repository contains code in multiple languages. The prioritization heuristics for language selection have the following order of preference:

1. The main language, if it belongs to the primary category;
2. The second most-used language, if in the primary category;
3. The main language, if in the secondary category;
4. The second most-used language, if in the secondary category;

This hierarchy reflects the assumption that primary languages are more likely to encapsulate the software's central functionality, while secondary languages contribute auxiliary capabilities.

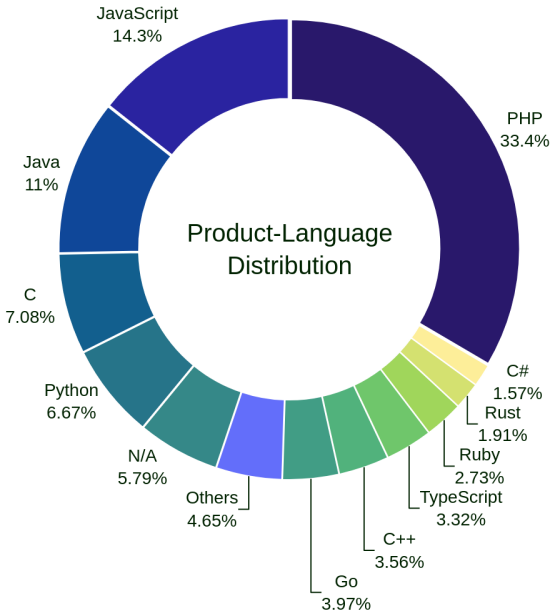


Figure 2.5: Distribution of Programming Language in the Labelled Set of Products

The integration of package metadata with repository-level analysis produces a dataset of 14,927 unique entries that maps software products to their primary implementation languages, covering a diverse set of 63 programming languages. Figure 2.5 illustrates that the resulting distribution of programming languages reflects prevailing trends in the open-source ecosystem. PHP emerges as the most common language, accounting for 33.4% of the mapped products, followed by JavaScript (14.3%), Java (11.0%), and Python (6.7%). Notably, 5.8% of the entries could not be confidently assigned a language and are marked as "N/A." Other frequently observed languages include Go (4.0%), C++ (3.6%), TypeScript (3.3%), Ruby (2.7%), Rust (1.9%), and C# (1.6%), with the remaining 4.6% distributed among a long tail of less common languages. This language-labeled

dataset forms the basis for subsequent analyses examining the intersection of programming language and vulnerability characteristics.

#### 2.4.2.8 Inferring Programming Languages from Vulnerability Descriptions

To enrich the dataset with more reliable language annotations and to better support downstream analysis, we incorporate a heuristic-based text analysis technique aimed at identifying programming languages directly from CVE descriptions. This step complements the existing product-language mapping by addressing cases where the product metadata alone may be ambiguous or unavailable.

The implemented approach scans CVE descriptions for filenames with known file extensions, drawing on a curated list of language extension mapping based on the previous programming language classification. By extracting these filenames and counting the frequency of their associated extensions, we determine the most probable language used in the context of the reported vulnerability. A conservative matching strategy ensures that only likely file references (e.g., `index.php`, `main.py`) are considered, while false positives such as domain names and non-code artifacts are filtered out using regular expressions and URL parsing logic.

#### 2.4.2.9 Dataset Consolidation

We infer the programming language directly from descriptions in 11,924 of the 78,620 CVEs, substantially enhancing both the coverage and specificity of language attribution. These inferred labels are then integrated with the previously constructed datasets, which include the CVE-ID, CWE-ID, and detailed product metadata such as vendor, software type, and package source. To associate each CVE with the most representative software product, we apply a scoring heuristic that prioritizes certain software categories, such as libraries over web applications, and gives preference to GitHub-hosted packages. The result is a unified dataset where each entry corresponds to a single vulnerability and is enriched with attributes including CVE ID, CWE ID, vendor, product, software type, and programming language. In total, the consolidated dataset comprises 61,654 CVEs, of which 41,007 are linked to products with known metadata.

To highlight common patterns, we visualize the co-occurrence relationships among software types, programming languages, and CWEs (Common Weakness Enumerations) using a Sankey diagram. This diagram captures the most frequent triplets observed in the dataset. For instance, the most prevalent link involves PHP-based extensions affected by CWE-79 (Cross-site Scripting), with 2,375 such instances. Web applications written in PHP are similarly dominant, particularly in connection with CWE-79 (2,206 instances) and CWE-89 (SQL Injection, 1,046 instances). Other prominent trends include C utilities vulnerable to buffer overflows (CWE-787, 283 instances), and Java libraries linked to CWE-79 (224 instances) and CWE-352 (Cross-Site Request Forgery). These patterns underscore the relevance of language and software-type context in shaping vulnerability profiles. This dataset is an initial effort and offers a foundation for empirical vulnerability analysis, enabling nuanced exploration of how implementation choices relate to security outcomes.

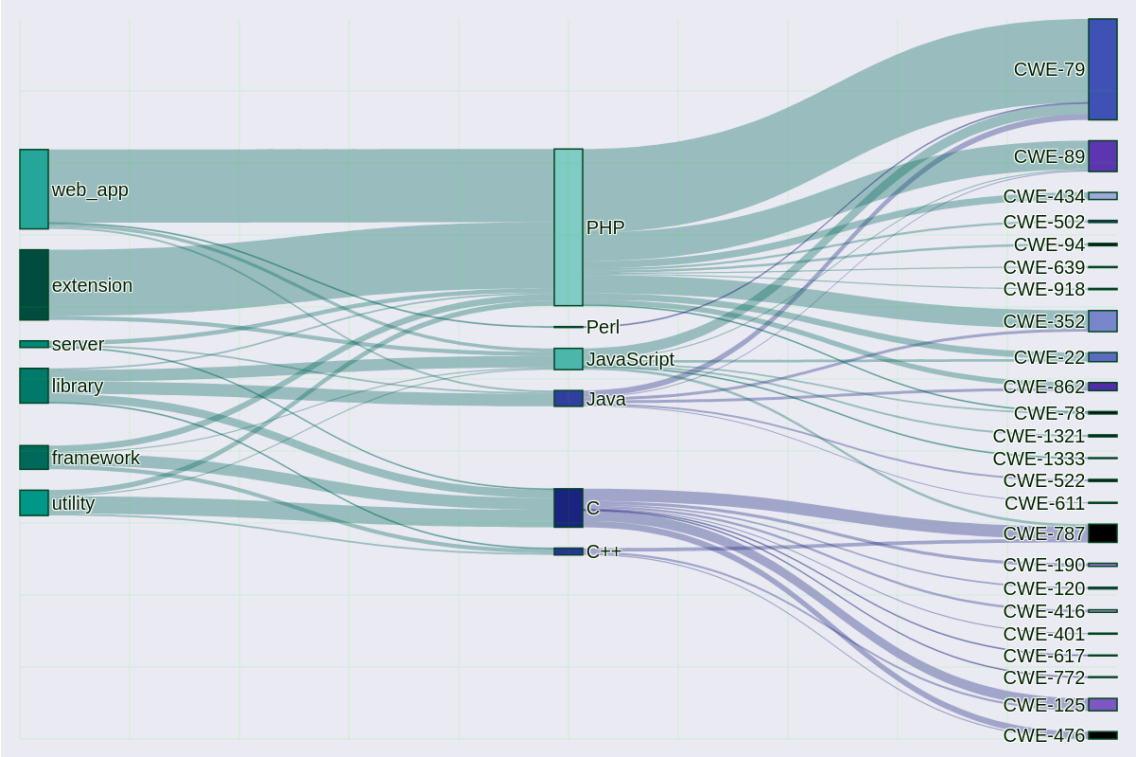


Figure 2.6: Sankey Diagram with the Relationships Among Attributes

### 2.4.3 Mapping Analysis Results to Classification Scheme

**Aim:** Analyze the collected data to identify recurring patterns and design a multi-level classification scheme for security faults. The goal is to move from raw data to insight, culminating in a hierarchical taxonomy (Dimension → Subclass → Example CWE) that reflects the observed diversity of vulnerabilities.

**Steps:**

- **Exploratory Analysis.** We perform statistical and visual analyses on the dataset. This includes generating histograms of vulnerabilities by language, technology, and layer; heatmaps showing co-occurrence of attributes; and scatter plots for correlations (e.g. severity vs. time). These visuals help highlight the most prominent fault attributes and gaps.
- **Pattern extraction.** From the visualizations we identify recurring themes. For instance, we may observe that web-related vulnerabilities (CWE-79, CWE-89) cluster in the "Technology: Web" category, or that certain root causes (e.g. improper input validation, CWE-20) appear across multiple languages and frameworks.
- **Classification hierarchy design.** Guided by the unified attributes (from Phase I) and patterns, we construct a three-level classification. The top level ("Dimension") corresponds

to our major analysis axes (e.g. *Programming Language*, *Technology Domain*, *Architectural Layer*, *Root Cause*). The second level ("Subclass") refines each dimension (e.g. for Language: *Web Languages* vs. *System Languages*). The third level lists concrete examples, typically specific CWE identifiers or vulnerability types (e.g. *CWE-79: XSS* under "Technology: Web"). An excerpt of this hierarchy is shown in Table 2.3, and its conceptual organization is visualized in Figure 2.7.

Table 2.3: Excerpt of the classification hierarchy (Phase III): Dimension → Subclass → Example CWE.

| **Dimension** | **Subclass** | **Example CWE** |
|---|---|---|
| Language | Web-facing languages | CWE-79 (Cross-site scripting) |
| Language | System languages | CWE-120 (Buffer overflow) |
| Technology | Web frameworks | CWE-89 (SQL injection) |
| Technology | Mobile applications | CWE-22 (Path traversal) |
| Layer | Network layer | CWE-319 (Cleartext transfer) |
| Layer | Application layer | CWE-416 (Use-after-free) |
| Root Cause | Input validation errors | CWE-20 (Improper input check) |
| Root Cause | Memory management errors | CWE-119 (Buffer overflow) |

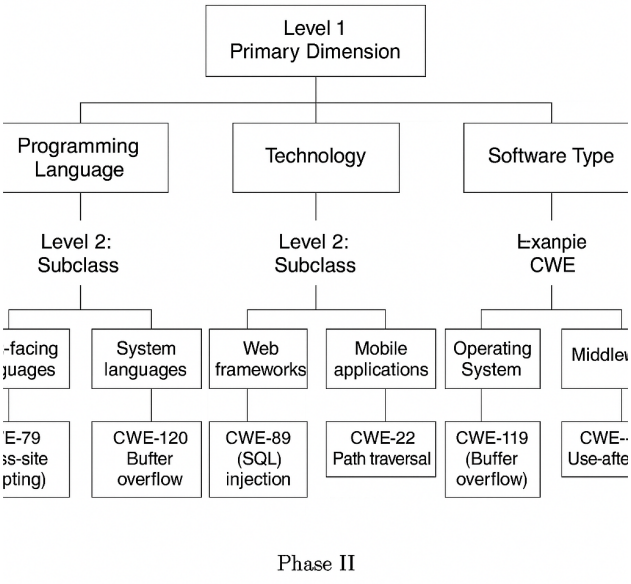


Figure 2.7: Hierarchical classification

Figure 2.7 (above) sketches the hierarchical classification derived in Phase III. Each Dimension branches into subclasses with illustrative CWE examples at the leaves. This taxonomy is rooted in the empirical data patterns uncovered and in prior taxonomies cited earlier. Through these steps, Phase III links the aggregated data back to our taxonomy framework, revealing how software vulnerabilities distribute across the multi-dimensional space. The resulting classification

scheme provides a structured way to profile security faults, based both on existing knowledge (the taxonomies) and new insights from the data analysis.

## 2.5 Results

Our analysis of the National Vulnerability Database (NVD) yielded a comprehensive dataset that allows us to profile security faults across multiple dimensions. Following the methodology outlined in Phase II, we present the results of our data collection and analysis, addressing the research question of how software security faults profile regarding software type, technology, causes, and programming language.

### 2.5.1 How do vulnerability patterns differ across software types?

To understand how vulnerability patterns vary by software type, we analyzed the 100%-stacked distribution of the top CWEs across seven categories of software products (extension, framework, library, mobile_app, server, utility, web_application). This analysis builds on the consolidated dataset of 55,657 CVEs for which software type was determined, after mapping products to types and programming languages as described in the methodology.

Figure 2.8 (100%-stacked bar chart) shows the relative frequency of four prominent CWEs (CWE-352: Cross-Site Request Forgery; CWE-787: Out-of-Bounds Write; CWE-79: Cross-Site Scripting; CWE-89: SQL Injection) along with "Others" (all remaining CWEs) for each software type. The key observations are as follows:

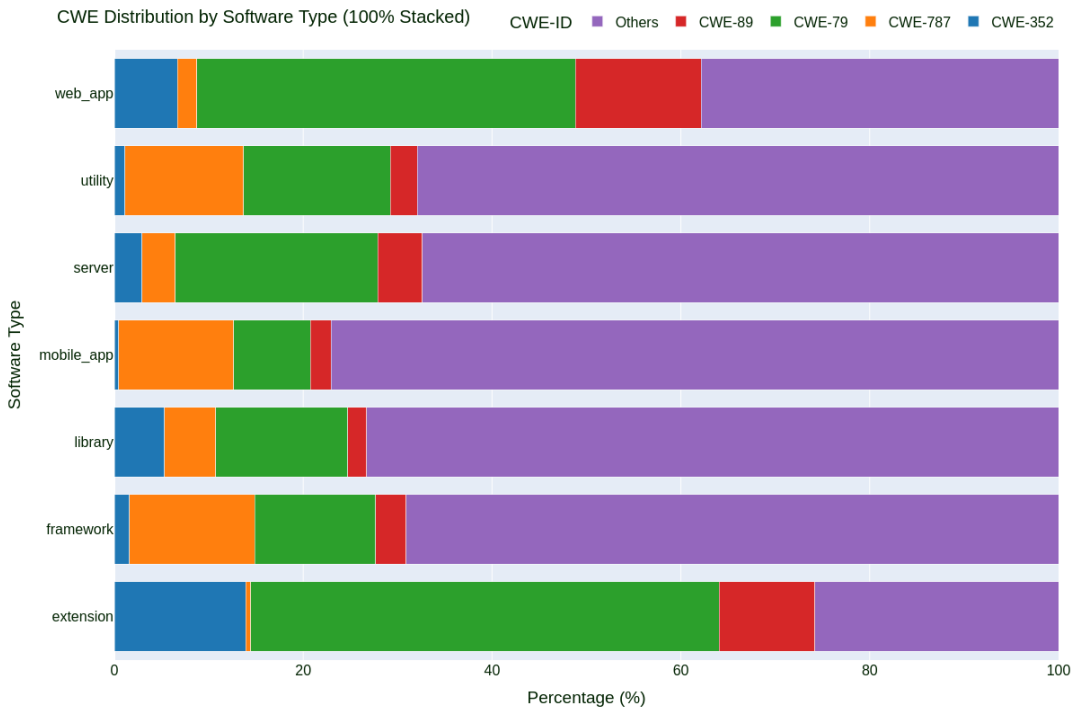


Figure 2.8: 100%-stacked bar chart of CWE Distribution by Software Type

- **Extensions**: Nearly half of the vulnerabilities in extensions are XSS (CWE-79: 49.6%), followed by CSRF (CWE-352: 13.9%) and SQL Injection (CWE-89: 10.2%), with a substantial residual portion (25.8%) in "Others." This suggests that browser or plugin-like components are predominantly exposed to web-based injection and script injection flaws, likely due to their frequent interaction with untrusted web content or APIs.

- **Web Applications**: Web applications similarly exhibit a high proportion of XSS (40.1%) and SQL Injection (13.3%), with CSRF accounting for 6.7% and "Others" comprising 37.9%. This aligns with expectations for web-facing services, where input validation and session/state management issues are primary concerns.

- **Servers**: Server software shows a moderate share of XSS (21.5%) and lower SQL Injection (4.7%) and CSRF (2.9%), with "Others" at 67.4%. The lower injection percentages may reflect that many server-side components rely on different interfaces (e.g., APIs, network protocols) where other types of weaknesses (captured in "Others") predominate.

- **Frameworks**: Frameworks stand out with a notable portion of memory-related issues (CWE-787: 13.3%), alongside XSS (12.8%) and a large "Others" category (69.1%). This indicates that underlying libraries or platforms (e.g., web frameworks, application frameworks) face both input-handling flaws and lower-level implementation bugs, including memory safety vulnerabilities.

- **Libraries**: Libraries have a smaller fraction of XSS (14.0%) and Out-of-Bounds Write (5.4%), with most vulnerabilities falling into "Others" (73.3%). This diffuse distribution suggests libraries span a wide range of functionality, so specific weakness patterns vary greatly depending on the domain (e.g., cryptography, data processing, image handling).

- **Utilities**: Utility software shows a moderate XSS share (15.6%) and a higher proportion of Out-of-Bounds Write (12.5%), with "Others" at 67.9%. Utilities (e.g., command-line tools, system utilities) often involve file or memory operations, which may explain the relative prominence of memory-related flaws alongside occasional script or injection issues when utilities process complex inputs.

- **Mobile Applications**: Mobile apps exhibit a small fraction of XSS (8.1%) but a notable Out-of-Bounds Write share (12.2%), with "Others" dominating (77.0%). This pattern reflects that while mobile apps may embed webviews (leading to some XSS), many vulnerabilities arise from platform-specific memory handling or misuse of APIs.

Overall, software types with strong web-facing components (extensions, web applications) show high proportions of injection and scripting weaknesses, whereas types closer to system-level or framework-level code (frameworks, utilities, mobile apps) surface more memory-related issues. Libraries cover a broad spectrum, resulting in a large "Others" bucket. Servers lie in between, with diverse vulnerabilities but fewer classical web injection flaws.

***Finding 1.*** *Vulnerability patterns vary significantly by software type: web-facing contexts (extensions, web applications) are dominated by script- and injection-related weaknesses (e.g., XSS, SQL Injection), while frameworks, utilities, and mobile applications exhibit higher rates of memory safety issues (e.g., out-of-bounds writes). Libraries show diffuse patterns across many weakness types. This implies that security mitigations should be tailored to software type, emphasizing input validation and sanitization for web components, and memory safety practices for frameworks and system-oriented software.*

### 2.5.2 How do programming languages correlate with security faults?

Our language mapping process successfully identified the primary programming language for 13,081 vulnerable software products. The distribution reveals several key insights:

- Web-oriented languages dominate the vulnerability landscape, with PHP (32.4%, 4,236 samples) and JavaScript (13.9%, 1,824 samples) accounting for nearly half of all identified vulnerabilities.
- Java (10.8%, 1,415 samples) represents a significant portion, reflecting its widespread use in enterprise applications.
- Systems programming languages like C (6.8%, 887 samples) and C++ (3.4%, 448 samples) show fewer vulnerabilities in absolute numbers but may represent more severe issues when they occur.
- Modern languages like Python (6.2%, 817 samples), Go (3.8%, 496 samples), and Rust (2.0%, 267 samples) are present but with lower frequency, potentially indicating better security properties or less widespread adoption.

The predominance of PHP and JavaScript vulnerabilities aligns with their extensive use in web development, where exposure to user input creates numerous attack vectors. The relatively lower representation of systems languages may reflect either better security practices or the challenges in vulnerability discovery for these languages.

### 2.5.3 Common Weakness Enumeration (CWE) Analysis

Our analysis extracted 18,566 CVE entries with associated CWE identifiers, revealing the most common vulnerability types:

- Cross-Site Scripting (CWE-79) dominates with 32.3% (6,000 samples) of all vulnerabilities, highlighting the persistent challenge of securing user input in web applications.
- Cross-Site Request Forgery (CWE-352, 7.7%, 1,434 samples) and SQL Injection (CWE-89, 7.1%, 1,316 samples) represent the next tier of common web vulnerabilities.

- Path Traversal (CWE-22, 4.8%, 899 samples) and Authorization Issues (CWE-862, 3.9%, 727 samples) round out the top five.
- Memory corruption vulnerabilities like Out-of-bounds Write (CWE-787, 3.6%, 661 samples) and Out-of-bounds Read (CWE-125, 2.3%, 422 samples) appear less frequently but often represent higher severity issues.

The prevalence of web-related vulnerabilities (CWE-79, CWE-352, CWE-89) reflects the extensive attack surface of web applications and the challenges in properly validating and sanitizing user input. Memory corruption vulnerabilities, while less common, remain a persistent threat, particularly in systems programming contexts.

### 2.5.4 Multi-dimensional Vulnerability Profiling

The consolidated dataset (18,566 samples total, including 7,868 products with both language and software type identified) reveals important patterns across software types, programming languages, and vulnerability classes:

- PHP-based extensions and web applications are particularly susceptible to Cross-Site Scripting (CWE-79), with 2,175 and 1,450 instances respectively, representing the most common vulnerability profiles.
- Cross-Site Request Forgery (CWE-352) is also prevalent in PHP extensions (607 instances) and web applications (334 instances).
- SQL Injection (CWE-89) affects PHP web applications (477 instances) and extensions (399 instances) most frequently.
- JavaScript applications show significant vulnerability to Cross-Site Scripting (CWE-79, 245 instances), while Java libraries exhibit both XSS (221 instances) and CSRF (188 instances) vulnerabilities.
- Memory corruption vulnerabilities cluster in C-based frameworks (CWE-787, 158 instances; CWE-125, 147 instances) and utilities (CWE-125, 154 instances; CWE-787, 138 instances).

These patterns reveal distinct vulnerability profiles across the software ecosystem:

1. **Web Application Profile:** Dominated by input validation vulnerabilities (XSS, CSRF, SQLi) in PHP and JavaScript applications, particularly in extensions and web applications.
2. **Systems Software Profile:** Characterized by memory corruption issues (buffer overflows, use-after-free) in C and C++ utilities and frameworks.
3. **Enterprise Application Profile:** Represented by a mix of web vulnerabilities and authorization issues in Java libraries and frameworks.

### 2.5.5 Addressing the Research Question

Returning to our research question—*How do software security faults profile regarding software type, technology, causes, and programming language?*—our analysis reveals several key insights:

- **Software Type:** Extensions and web applications are most vulnerable to input validation issues, while utilities and frameworks show greater susceptibility to memory corruption. Servers exhibit a more diverse vulnerability profile spanning both categories.
- **Technology:** Web technologies dominate the vulnerability landscape, with client-side (XSS, CSRF) and server-side (SQLi, path traversal) issues being most prevalent.
- **Causes:** Improper input validation emerges as the dominant root cause across the ecosystem, followed by memory management issues in systems software and authorization problems across multiple software types.
- **Programming Language:** Strong correlations exist between languages and vulnerability types: PHP and JavaScript with web vulnerabilities, C and C++ with memory corruption, and Java with a mix of web and authorization issues.

These findings demonstrate that vulnerability profiles are not uniform across the software ecosystem but rather cluster into distinct patterns based on software type, implementation language, and application domain. This multi-dimensional profiling provides a more nuanced understanding of security fault distribution than previous single-dimensional analyses.

## 2.6 Discussion

Our multi-dimensional analysis of security faults provides a comprehensive framework for understanding vulnerability patterns across different programming languages, software types, and architectural layers. The hierarchical classification scheme developed through our methodology offers a structured approach to categorizing and analyzing security vulnerabilities, which can inform both research and practice in software security.

### 2.6.1 Limitations

While our analysis provides valuable insights, several limitations should be acknowledged:

- **Data source limitations:** The reliance on NVD data means our analysis is limited to reported and publicly disclosed vulnerabilities, which may not represent the complete vulnerability landscape. Vulnerabilities that are discovered but not reported, or those that remain undiscovered, are not captured in our dataset.
- **Language mapping challenges:** The process of mapping software products to programming languages relies on heuristics and predefined mappings. For repositories with multiple

languages, our priority-based selection may not always capture the language most relevant to the vulnerability.

- **Software type categorization:** The categorization of software into different types (e.g., extension, package, framework) is based on predefined mappings and keyword analysis, which may not always accurately reflect the true nature of the software.
- **CWE selection:** Our methodology selects the "most appropriate" CWE ID based on vulnerability mapping, abstraction level, and weakness type. This selection process may not always capture the full complexity of vulnerabilities that span multiple weakness types.
- **Temporal limitations:** Our analysis represents a snapshot of the vulnerability landscape at the time of data collection. The distribution and patterns of vulnerabilities may change over time as new technologies emerge and security practices evolve.
- **Abstraction trade-offs:** The hierarchical classification scheme, while providing a structured approach to categorizing vulnerabilities, necessarily involves some level of abstraction that may obscure nuanced differences between similar vulnerability types.

Despite these limitations, the large sample sizes and clear patterns observed provide confidence in the overall trends identified in our analysis. The multi-dimensional approach allows for a more comprehensive understanding of security faults than would be possible with a single-dimensional analysis.

## 2.7 Threats to Validity

In this section, we discuss potential threats to the validity of our study, categorized into four main types: internal validity, external validity, construct validity, and conclusion validity.

### 2.7.1 Internal Validity

Internal validity concerns the extent to which our methodology allows for accurate causal inferences.

- **Data extraction accuracy:** The automated data extraction process may introduce errors or inconsistencies. While we implemented validation checks, the complexity of parsing and merging data from multiple sources could lead to inaccuracies in the final dataset.
- **Selection bias in CWE mapping:** Our algorithm for selecting the "most appropriate" CWE ID for each vulnerability involves prioritization based on vulnerability mapping, abstraction level, and weakness type. This selection process may introduce bias by favoring certain types of weaknesses over others.

- **Temporal effects:** The NVD database is continuously updated, with both new vulnerabilities being added and existing entries being modified. Our analysis represents a snapshot at a specific point in time, which may affect the stability of our findings.

### 2.7.2 External Validity

External validity concerns the generalizability of our findings to other contexts.

- **Representativeness of the sample:** While the NVD is a comprehensive database, it may over-represent certain types of software or vulnerabilities. For example, open-source projects may be more likely to have their vulnerabilities reported and documented compared to proprietary software.
- **Technological evolution:** The rapid evolution of technology means that the patterns and distributions of vulnerabilities observed in our study may not generalize to future software systems or emerging technologies.

### 2.7.3 Construct Validity

Construct validity concerns whether we are measuring what we intend to measure.

- **Language classification:** Our classification of programming languages into primary (general-purpose) and secondary (domain-specific) categories may not capture the full complexity of language ecosystems. Some languages may span multiple categories or evolve over time.
- **Software type categorization:** The categorization of software into different types based on predefined mappings and keyword analysis may not always accurately reflect the true nature or purpose of the software.
- **CWE abstraction levels:** The CWE hierarchy includes different levels of abstraction (e.g., Class, Base, Variant). Our preference for more specific abstractions may not always align with how vulnerabilities are conceptualized in practice.

### 2.7.4 Conclusion Validity

Conclusion validity concerns the reliability of our conclusions based on the data and analysis.

- **Pattern identification:** The identification of patterns in the exploratory analysis phase relies on visual inspection and statistical methods. There is a risk of identifying spurious patterns or missing significant but subtle relationships.
- **Classification hierarchy design:** The design of our three-level classification hierarchy (Dimension $\rightarrow$ Subclass $\rightarrow$ Example CWE) involves subjective decisions about how to organize and categorize vulnerabilities. Different researchers might arrive at different classification schemes given the same data.

- **Interpretation of results:** The interpretation of the relationships between software types, programming languages, and CWEs may be influenced by preconceptions or biases in how we understand and categorize security vulnerabilities.

To mitigate these threats, we have implemented several measures, including validation checks in our data extraction pipeline, cross-referencing multiple data sources, and using established taxonomies as a foundation for our classification scheme. We also acknowledge that our findings should be interpreted within the context of these limitations and that further research is needed to validate and extend our results.

## 2.8 Summary

**Eduard:** TODO END

# References


[1] J. Hughes and G. V. Cybenko, "Quantitative metrics and risk assessment: The three tenets model of cybersecurity," *Technology Innovation Management Review*, vol. 3, pp. 15–24, 2013.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. DOI: 10.1109/TDSC.2004.2.

[3] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[4] N. I. of Standards and Technology, *NIST Glossary*, Accessed 7-June-2022, Jun. 2022. [Online]. Available: https://csrc.nist.gov/glossary.

[5] G. McGraw, "Software security," *Datenschutz und Datensicherheit - DuD*, vol. 36, pp. 662–665, 2004.

[6] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," vol. 26, no. 3, pp. 211–254, Sep. 1994, ISSN: 0360-0300. DOI: 10.1145/185403.185412. [Online]. Available: https://doi.org/10.1145/185403.185412.

[7] J. Zwolak, *Program Bug Examples*, Accessed 21-October-2020, Oct. 2000. [Online]. Available: http://courses.cs.vt.edu/~cs1206/Fall00/bugs_CAS.html.

[8] S. Castro, *Off-by-one overflow explained*, Accessed 21-October-2020, Oct. 2016. [Online]. Available: https://csl.com.co/en/off-by-one-explained/.

[9] A. A. Khwaja, M. Murtaza, and H. F. Ahmed, "A security feature framework for programming languages to minimize application layer vulnerabilities," *Security and Privacy*, vol. 3, 2020.

[10] S. R. Tate, M. Bollinadi, and J. Moore, "Characterizing vulnerabilities in a major linux distribution," in *SEKE*, 2020.

[11] C. Joshi, U. K. Singh, and K. Tarey, "A review on taxonomies of attacks and vulnerability in computer and network system," 2015.

[12] A. Bazaz and J. D. Arthur, "Towards a taxonomy of vulnerabilities," *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 163a–163a, 2007.

[13] F. Piessens, "A taxonomy of causes of software vulnerabilities in internet software," 2002. [Online]. Available: https://www2.cs.kuleuven.be/publicaties/rapporten/cw/CW346.pdf.

[14] S. Weber, P. A. Karger, and A. M. Paradkar, "A software flaw taxonomy," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–7, 2005.

[15] K. Goseva-Popstojanova and J. Tyo, "Security vulnerability profiles of mission critical software: Empirical analysis of security related bug reports," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:955555.

[16] S. Garg, R. K. Singh, and A. K. Mohapatra, "Analysis of software vulnerability classification based on different technical parameters," *Information Security Journal: A Global Perspective*, vol. 28, pp. 1–19, 2019.

[17] O. Ezenwoye, Y. Liu, and W. G. Patten, "Classifying common security vulnerabilities by software type," in *SEKE*, 2020.

[18] O. Ezenwoye and Y. Liu, "Web application weakness ontology based on vulnerability data," *ArXiv*, vol. abs/2209.08067, 2022.

[19] M. Corporation, *CWE List*, Accessed 31-May-2025, Jun. 2025. [Online]. Available: https://cwe.mitre.org/data/index.html.

[20] N. Mansourov and D. Campara, *System Assurance: Beyond Detecting Vulnerabilities*, First. Morgan Kaufmann, 2010, ISBN: 9780123814159.

[21] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, 11:1–11:46, 2012.