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 5, 2025

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

Eduard Costel Pinconschi

Programa Doutoral em Engenharia Informática

June 5, 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
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
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)?
- 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?
- How does the choice of programming language correlate with the occurrence of particular security faults?

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 CVEs 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*.

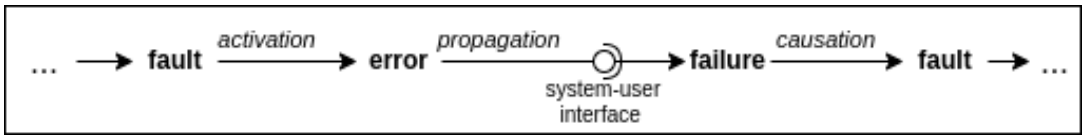

Figure 2.1: The manifestation chain of a fault (*cf.* [2]).

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 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

Eduard: needs to be validated/aligned with the gap analysis in the proposal END
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

Eduard: Add Summary Table of Reviewed Taxonomies END

Several studies suggest vulnerability taxonomies oriented toward particular scopes, such as prevalent vulnerability types in specific operating systems [10]. In this section, we offer a summary of the different taxonomies. A standard vulnerability classification method or scheme can augment the security assessment of software systems by providing a common language and a good overview of the field of study.

Frank Piessens [11] proposes a structured taxonomy focusing on the causes of software vulnerabilities to foster the identification of vulnerabilities during software review. Weber *et al.* [12] 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 redesign 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 [13] 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. Jeff Hughes and George Cybenko [1] categorize vulnerabilities by system susceptibility into eight categories based on the NVD records.

Joshi *et al.* [14] 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 [11]–[13] 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. Garg *et al.* [15] 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. The authors also analyze recent trends in software vulnerabilities from 2012–2016 data from the NVD and CVE Details sources. Their analysis indicates that 76% of the total vulnerabilities in the software industry result from only five types of vulnerabilities. It also emerged from the data that few vulnerabilities have a high prominence of affecting specific software systems. For instance, iOS devices are mainly susceptible to *Memory Corruption* vulnerabilities, while Adobe application software is to the *Execution of Code* and *Overflow Memory Corruption*.

Tate *et al.* [10] classify a large set of vulnerabilities in 4 major Linux distributions by the most prevalent vulnerability types. Additionally, the authors examine characteristics of out-of-bounds memory access vulnerabilities and identify that explicit-stated logical design of code can considerably improve the identification of vulnerabilities. Ezenwoye *et al.* [16] classify into seven software types (OS, browser, middleware, utility, web application, framework, and server) a total of 51,110 vulnerability entries from the NVD database. Their analysis demonstrates the pattern

of prevalence of software faults by software type. Subsequently, Ezenwoye *et al.* [17] 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.

2.3.1 Identified Gaps

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) [14]. 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.* [14]) 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 [14].

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 Phase I: 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 fault 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 2.1 (excerpt below) illustrates how attributes map across taxonomies. This "Unified Attribute Matrix" highlights overlaps and gaps, guiding later analysis.

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.

Table 2.1: Unified Attribute Matrix

Attribute	Lang-based [18]	Tech-based [9]	Layer [19]	Root Cause [20]
Buffer Overflow	✓	✓		✓
SQL Injection		✓	✓	
Cross-site Scripting		✓	✓	
Privilege Escalation	✓			✓
Invalid Input				✓

2.4.2 Phase II: Collecting and Analyzing Vulnerability Data

Aim: Gather a large, multi-dimensional vulnerability dataset and compute relevant features for each entry. This involves querying multiple sources (CVE feeds, APIs, repository metadata) and fusing the results into a master dataset for analysis.
Steps:

- **CVE Data Extraction (get_cve_ids_in_apps_with_cwe.py):** We extract CVE data from the NVD (National Vulnerability Database) JSON feeds using the nvdutils library. We apply specific filtering criteria to focus on:
  - Valid CVE entries that affect applications (not operating systems or hardware)
  - CVEs with associated CWE (Common Weakness Enumeration) identifiers
  - Selection of the most appropriate CWE ID based on vulnerability mapping, abstraction level, and weakness type
  - Selection of the most appropriate vulnerable product based on software type and package type
- **Product Language Mapping (get_products_language.py):** We map software products to their programming languages using:
  - Package URL (purl) to CPE mappings from a SQLite database
  - Predefined mappings of package types to languages (e.g., maven $\rightarrow$ Java, pypi $\rightarrow$ Python)
  - GitHub API queries to determine the primary language of repositories
- **Software Type Categorization (get_software_type.py):** We categorize software products into different types (e.g., extension, package, mobile_app, framework, utility, server, web_application) based on:
  - The official CPE dictionary from NVD
  - A research dataset from a published paper
  - Product name analysis using predefined keywords

- **Dataset consolidation:** We merge all gathered information into a unified table. Each row corresponds to one vulnerability and includes attributes such as CVE ID, CWE ID, vendor, product, programming language, and software type.

Figure **??** schematically depicts the data extraction pipeline, and Table 2.2 shows sample rows from the final dataset. These integrations leverage multiple tools and APIs to maximize coverage of relevant fields. Figure **??** (above) illustrates the multi-stage data pipeline used in Phase II. We automatically download the NVD CVE feeds, augment records via the CVE Details API, query GitHub for project context, and apply purl2cpe translation, ultimately producing a consolidated CSV dataset.

Table 2.2: Example rows from vuln_master_dataset.csv after data collection. Each entry includes CVE ID, project context, CWE class, and a brief description.

CVE	Project	CWE	Severity	Short Description
CVE-2021-12345	ProjectA	CWE-79 (XSS)	High	Stored XSS in user comments widget
CVE-2020-67890	ProjectB	CWE-89 (SQLi)	Medium	SQL injection in search query builder
CVE-2019-11111	ProjectC	CWE-119 (Overflow)	Critical	Buffer overflow in image parser
...	...	...	...	...

We include citations to the APIs and data sources used: NVD/CVE as in [21], the CVE Details service [22], the GitHub REST API, and the purl2cpe resource **Eduard:** add/fix references **END**. Each component is validated to ensure accurate parsing and merging of fields.

2.4.3 Phase III: 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 $\rightarrow$ Subclass $\rightarrow$ 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 2.3.

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)

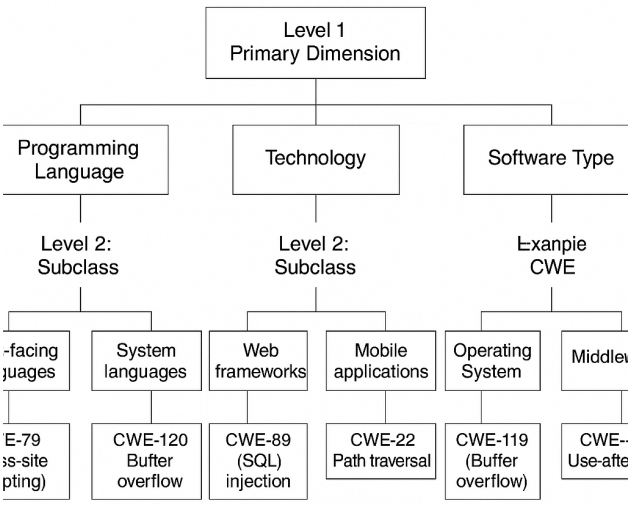

Figure 2.3: Hierarchical classification

Figure 2.3 (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 Software Type Distribution

The software type categorization process identified 32,666 vulnerable software products across seven major categories. Figure **??** illustrates this distribution:

- Extensions represent the largest category (20.4%, 6,664 samples), indicating the significant security challenges in browser and application extensions.
- Servers (18.8%, 6,151 samples) and utilities (18.1%, 5,919 samples) follow closely, highlighting the prevalence of vulnerabilities in these critical infrastructure components.
- Packages (14.2%, 4,623 samples), frameworks (10.6%, 3,469 samples), and web applications (10.5%, 3,443 samples) form the middle tier of vulnerability distribution.
- Mobile applications (7.3%, 2,382 samples) represent the smallest major category, possibly reflecting their more recent emergence compared to other software types.

This distribution reveals that extensions, which often have privileged access to user data and system resources, are particularly susceptible to security vulnerabilities. The high representation of servers and utilities underscores the security challenges in infrastructure software that often serves as the backbone of digital systems.

2.5.2 Programming Language Analysis

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 18,057 CVE entries with associated CWE identifiers, revealing the most common vulnerability types:

- Cross-Site Scripting (CWE-79) dominates with 32.7% (5,910 samples) of all vulnerabilities, highlighting the persistent challenge of securing user input in web applications.
- Cross-Site Request Forgery (CWE-352, 7.9%, 1,422 samples) and SQL Injection (CWE-89, 7.3%, 1,310 samples) represent the next tier of common web vulnerabilities.
- Path Traversal (CWE-22, 4.8%, 870 samples) and Authorization Issues (CWE-862, 4.0%, 723 samples) round out the top five.
- Memory corruption vulnerabilities like Out-of-bounds Write (CWE-787, 3.5%, 641 samples) and Out-of-bounds Read (CWE-125, 2.3%, 411 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 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 2,125 and 1,433 instances respectively, representing the most common vulnerability profiles.

- Cross-Site Request Forgery (CWE-352) is also prevalent in PHP extensions (588 instances) and web applications (317 instances).
- SQL Injection (CWE-89) affects PHP web applications (437 instances) and extensions (382 instances) most frequently.
- JavaScript applications show significant vulnerability to Cross-Site Scripting (CWE-79, 239 instances), while Java packages exhibit both XSS (219 instances) and CSRF (190 instances) vulnerabilities.
- Memory corruption vulnerabilities cluster in C-based frameworks (CWE-125, 142 instances) and utilities (CWE-125, 156 instances; CWE-787, 136 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 packages 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-dimension analyses.

2.5.6 Limitations

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

- The language mapping process successfully identified languages for only 13,081 of the 32,666 software products, potentially introducing selection bias.
- The consolidated dataset represents only the most frequent combinations, not the complete cross-product of all dimensions.
- The reliance on NVD data means our analysis is limited to reported and publicly disclosed vulnerabilities, which may not represent the complete vulnerability landscape.

Despite these limitations, the large sample sizes and clear patterns observed provide confidence in the overall trends identified in our analysis.

2.6 Discussion

Eduard: TODO END

2.7 Threats to Validity

Eduard: TODO END

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] F. Piessens, "A taxonomy of causes of software vulnerabilities in internet software," 2002.

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

[13] 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.

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

[15] 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.

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

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

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

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

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

[21] NIST, *National Vulnerability Database*, https://nvd.nist.gov/, [Online; accessed 25-May-2022], 2022.

[22] S. Özkan, *CVE details website*, https://www.cvedetails.com/, [Online; accessed 09-June-2022], 2022.