# DOMAIN-SPECIFIC LLMS-BASED REFACTORING TECHNIQUES FOR SOFTWARE VULNERABILITIES

by

OBIEDA MOHAMMAD SALEM ANANBEH

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE AND INFORMATICS

2025

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Dae-Kyoo Kim, Ph.D., Chair
Lunjin Lu, Ph.D.
Li Li, Ph.D.
Hua Ming, Ph.D.

*This dissertation is dedicated to all members of my family. I would like to thank everyone who has stood by me during this academic pursuit.*

# ACKNOWLEDGMENTS

I would like to thank my family for their unwavering support. I have been sustained throughout my life by their encouragement, love, and belief in me. This accomplishment would not have been possible without their constant presence and understanding. This work is also dedicated to everyone who has played a positive role in my life, offering kindness, inspiration, or a helping hand along the way.

Additionally, I would like to thank my supervisor, Professor Dae-Kyoo Kim, for his patience, insightful guidance, and continuous support throughout my doctoral journey. His mentorship has been instrumental in shaping both this dissertation and my development as a researcher.

It was a pleasure working with the committee members, including Professors Lunjin Lu, Li Li, and Hua Ming. I am grateful for their valuable time, thoughtful feedback, and encouragement, which strengthened this work and motivated me to aim higher.

Obieda Mohammad Salem Ananbeh

# ABSTRACT

DOMAIN-SPECIFIC LLMS-BASED REFACTORING TECHNIQUES FOR SOFTWARE VULNERABILITIES

by

Obieda Mohammad Salem Ananbeh

Adviser: Dae-Kyoo Kim, Ph.D.

Software vulnerabilities pose a significant threat to the security of systems. While much of the existing research focuses on detecting vulnerabilities, the potential of refactoring for vulnerability mitigation has not been explored much. To bridge this gap, this work introduces a set of refactoring techniques aimed at mitigating various types of vulnerabilities.

The process begins by categorizing vulnerabilities based on the weakness types defined in the CWE system, concentrating on eight categories that pose significant risks. For each CWE within each category, five samples are generated using ChatGPT, resulting in a dataset of 405 samples. These samples are rigorously analyzed manually for their validity. Based on the dataset, the characteristics of each category are identified, the core problem within the category is defined, and a specific refactoring solution is developed to address it. These techniques are evaluated using the Snyk tool on twenty-one active open-source projects. The results demonstrate an 89% reduction in vulnerabilities after applying the refactoring techniques, providing insights on enhancing software security through refactoring-based strategies.

Building upon this foundation, this work introduces VulnFixAI, an automated tool designed to detect and repair various types of vulnerabilities in Java code by integrating refactoring techniques with fine-tuned domain-specific large language models (DSLLMs).

Specifically, VulnFixAI implements three targeted refactoring algorithms, Whitelist Validation Refactoring (WVR), Output Safety Refactoring (OSR), and TrustChain Verification Refactoring (TCVR), which were selected for their demonstrated effectiveness against prevalent CWE vulnerabilities identified in the initial study. A dataset of 10,000 vulnerable Java snippets was collected, extracted from 907 open-source, and fine-tuned Llama 3.2 (3B parameters) model to enhance vulnerability detection and automated repair. VulnFixAI was evaluated on 20 real-world open-source projects listed in the GitHub Advisory Database. The results demonstrate an 89% overall effectiveness in detecting and repairing vulnerabilities, representing a 51% improvement over non-fine-tuned Llama 3.2 (3B) and an average 27% improvement over other LLMs, including ChatGPT 4, Claude 3.5 Sonnet, and Gemini 2.0 Flash. These findings underscore how VulnFixAI's hybrid approach, integrating fine-tuned DSLLMs and refactoring algorithms, provides an efficient, scalable, and highly effective solution for enhancing software security.

This work introduces eight novel refactoring techniques specifically designed to address the root causes of vulnerabilities and presents VulnFixAI, an automated tool that integrates these techniques with DSLLMs to detect and repair vulnerabilities with high precision. Evaluated on 20 real-world projects, VulnFixAI achieved an overall effectiveness rate of 89%, significantly outperforming leading LLMs such as ChatGPT-4 74%, Claude 3.5 Sonnet 70%, Gemini 2.0 Flash 67%, and non-fine-tuned Llama 3.2 59%. These results demonstrate VulnFixAI's superior ability to identify and repair vulnerabilities through domain-specific tuning and structured refactoring.

TABLE OF CONTENTS

TABLE OF CONTENTS—Continued

LIST OF TABLES

LIST OF FIGURES

# CHAPTER ONE

# INTRODUCTION

The National Institute of Standards and Technology (NIST) estimates that security bugs and vulnerabilities cost the US economy approximately $60 billion annually [79]. This figure includes the costs associated with implementing patches to address security issues, as well as the direct impacts of the vulnerabilities themselves [31]. The presence and exploitability of these vulnerabilities are heavily influenced by the design and implementation choices of a system [107]. In addition, third-party code and libraries frequently contain vulnerabilities that may be unintentionally incorporated into a software system, thus elevating its overall security risk [59]. Research on vulnerabilities and security bugs consistently emphasizes the correlation between poor code quality and the prevalence of vulnerabilities [58, 93]. The Common Weakness Enumeration (CWE) [29] provides a framework for classifying types of software weaknesses, which aids in the identification, mitigation, and prevention of vulnerabilities in software systems. Further research has developed metrics and methodologies for assessing the severity of various CWEs, discussing their impact on software security [99].

Existing research has primarily focused on vulnerability detection and patching, yet the potential of refactoring for vulnerability mitigation has remained largely unexplored [88]. To address this gap, this work presents a set of refactoring techniques aimed at addressing vulnerabilities at their root cause. Although there is extensive literature on general source code refactoring (e.g., [14, 15, 130, 142]), efforts specifically aimed at refactoring for vulnerability mitigation are sparse.

Traditionally, the task of fixing software vulnerabilities has been performed manually during code reviews where developers rely on their experience to identify insecure

patterns and apply appropriate fixes. [149, 151]. While manual analysis allows for expert scrutiny, it is time-consuming, resource-intensive, and heavily dependent on specialized security expertise, which is often in short supply [12, 83]. To address these limitations, researchers have explored automated techniques such as static and dynamic analysis tools [34, 64, 94, 95, 121, 164, 165],they do not directly assist in applying consistent, reliable fixes. Static analysis identifies security flaws at the code level without execution, whereas dynamic analysis detects vulnerabilities during runtime testing. However, both methods suffer from challenges such as high false-positive rates, limited scalability, and difficulty in adapting to new types of vulnerabilities [17, 123, 132].

VulnFixAI differs from traditional approaches by combining fine-tuned domain-specific large language models (DSLLMs), with structured refactoring algorithms and supporting a significantly broader range of CWE types.

Recent advancements in Artificial Intelligence (AI), particularly in Machine Learning (ML) and Deep Learning (DL), have significantly improved software vulnerability Fixing [2, 21, 49, 88, 97, 128, 148, 150]. The ability of ML/DL models to analyze source code and capture complex patterns has led to significant improvements over traditional fixing techniques [22, 62, 154].

Recent advancements in vulnerability fixing have led to the development of a variety of automated tools and techniques. Early efforts such as SemFix [104] and SketchFix [66] represent symbolic and generate-and-validate approaches that synthesize semantically correct patches through program analysis and constraint solving. More recent efforts have adopted learning-based techniques. For example, DeepCode AI Fix [16] applies prompt-tuned LLMs with security-focused fine-tuning to fix over 80% of reported vulnerabilities, while TemVUR [86] leverages template-based repair for Java binaries, outperforming source-level methods on Vul4J [20]. Multi-granularity detectors [105] and graph-based models such as ANGLE [120] enhance fix accuracy by capturing seman-

tic and structural patterns in code. CleanVul [84] combines LLM heuristics with static analysis to improve the labeling of vulnerable code, and Colefunda [163] uses contrastive learning to explain and detect covert security fixes. Other innovative directions include RL-guided repair [73], which rewards secure and compilable edits, and chain-of-thought prompting [108], which enhances fix interpretability and accuracy. NAVRepair [153] uses AST-based analysis to guide vulnerability repair in C/C++ by generating repair prompts tailored to specific error types. SPDetect [67] employs symbolic execution and safety properties for detecting vulnerabilities in C/C++. SedSVD [35] and HLT [25] are deep learning models trained exclusively on C/C++ datasets, while TestInv-Code [96] uses dynamic analysis and passive testing for C code vulnerability detection.

While these approaches demonstrate impressive capabilities, most either focus on C/C++ vulnerabilities limiting their applicability to Java-based systems, or target a limited set of CWE types, constraining the scope of their analysis. Although many rely on pattern learning, they often lack security-oriented refactoring techniques. In contrast, this work introduces a set of refactoring techniques designed to address vulnerabilities at their root cause. These techniques are integrated into VulnFixAI, a hybrid framework that combines structured refactoring techniques with fine-tuned DSLLMs to enable automated, CWE-aligned, and explainable vulnerability fixing in Java code.

More recently, Large Language Models (LLMs), such as OpenAI's GPT family [78], Meta's Llama [145], Google's Gemini [139], and Anthropic's Claude [7], have demonstrated remarkable capabilities in code generation, completion, and vulnerability detection. Trained on massive datasets comprising text and code, LLMs can learn intricate representations of programming semantics, enabling them to assist in vulnerability analysis [53, 82, 143]. However, despite their potential, LLM-based vulnerability detection introduces several challenges, including concerns about data privacy [24], vendor

lock-in, limited model control [136], unpredictable performance, and high operational costs, particularly when processing large codebases or handling frequent queries [23, 77].

To the best of current knowledge, this study is the first to specifically tackle refactoring for vulnerability mitigation. Vulnerabilities are categorized based on weakness types defined in the CWE system, focusing on eight categories that pose significant risks. For each CWE in each category, five samples are generated using ChatGPT [113], resulting in a dataset of 405 samples. These samples are rigorously analyzed manually to confirm their validity. Based on the dataset, common characteristics of each category are identified, the specific problem the category presents is defined, and a tailored refactoring solution is developed for each problem. These techniques are evaluated using the Snyk tool [133], which detects vulnerabilities in software systems, across twenty-one active open-source projects spanning various domains.

Building on this foundation, this work introduces VulnFixAI, an automated tool designed to bridge the gap between vulnerability detection and structured remediation by applying refactoring techniques that address the root causes of vulnerabilities. In this work, The refactoring techniques WVR, OSR, and TCVR are adopted for their demonstrated effectiveness against prevalent CWE vulnerabilities identified in the first study, transforming them into automated algorithms within VulnFixAI. To enhance both detection accuracy and repair robustness, these techniques are integrated with fine-tuned DSLLMs, specifically Llama 3.2 (3B parameters), leveraging AI-driven pattern recognition to systematically guide vulnerability mitigation.

The proposed approach leverages a curated dataset of 10,000 vulnerable Java code snippets extracted from 907 open-source projects spanning domains such as Android and web applications. VulnFixAI's effectiveness is evaluated on validated vulnerabilities from 20 real-world Java projects selected from the GitHub Advisory Database.

Specifically, this work aims to address the following research questions.

RQ1: *What are the different types of software vulnerabilities categorized by their root causes?* While the CWE system provides a broad categorization, it lacks the granularity needed to pinpoint specific issues directly. this work aim to develop more fine-grained categories that capture the essential characteristics of the problems, enabling the formulation of targeted refactoring solutions.

RQ2: *What strategies should be employed to effectively address the vulnerabilities of these categories?* Once vulnerabilities are categorized, it becomes crucial to investigate how their root causes can be effectively tackled through specific refactoring strategies based on the characteristics of each category.

RQ3: *Is there any interdependence among the proposed refactoring techniques?* Given that vulnerabilities often lead to one another, it is important to understand whether refactoring techniques also exhibit dependencies that should be considered when applying them.

RQ4: *How effective are the proposed refactoring techniques in addressing targeted vulnerabilities?* The proposed refactoring techniques need to be evaluated for their effectiveness. This evaluation should be carried out on real-world projects to ensure the practicality of the techniques.

RQ5: *To what extent does fine-tuning DSLLMs on a domain-specific dataset improve the accuracy of vulnerability detection and automated fixes compared to non-fine-tuned DSLLMs?* While DSLLMs like Llama 3.2 (3B) provide a lightweight alternative to large LLMs, their general purpose lacks deep security-specific understanding. By fine-tuning Llama 3.2 (3B) on a domain-specific dataset of vulnerable Java code, this work aims to assess improvements in detection precision, fix accuracy, and overall repair effectiveness compared to the non-fine-tuned version of the same model.

RQ6: *Can a hybrid approach combining fine-tuned DSLLMs and refactoring algorithms enhance the automated detection and repair of software vulnerabilities compared to existing state-of-the-art models?* While LLMs excel in vulnerability detection, their repair capabilities often lack structure and consistency. By integrating fine-tuned DSLLMs with systematic refactoring techniques, this work seeks to evaluate whether this hybrid approach outperforms leading AI-based solutions, such as ChatGPT-4, Claude 3.5, and Gemini 2.0 Flash, in terms of accuracy and robustness of security fixes.

RQ7: *How do fine-tuned DSLLMs perform across different categorized vulnerabilities compared to LLMs?* Different software vulnerabilities present unique challenges with some being easier to detect and repair than others. this work aims to investigate how fine-tuned DSLLMs perform across multiple vulnerability categories compared to general LLMs. Specifically, the assessment focuses on how these models handle Lack of Whitelist Validation (LOWV), Lack of Input Sanitization (LOIS), and Inadequate TrustChain Verification (ITV) regarding their repair success rates and overall security robustness.

# CHAPTER TWO

# RELATED WORK

There exists much work exploring various aspects of refactoring in software security (e.g., [1, 3, 4, 5, 6, 11, 26, 36, 40, 70, 71, 72, 76, 98, 101, 111, 156]).

To contextualize this contribution, the related work is organized into three areas: (1) the impact of refactoring on software security, (2) traditional vulnerability detection and repair techniques, and (3) machine learning and deep learning-based approaches.

### 2.0.1 Refactoring and Its Impact on Software Security

Abid et al. [1] investigated the impact of general source code refactoring on software security alongside other quality objectives. They discussed the trade-offs developers face between maintaining security and enhancing other quality attributes such as reusability, proposing a multi-objective refactoring recommendation approach. This study, which was evaluated on 30 open-source projects, found that quality improvements through refactoring often negatively correlate with security, echoing the findings of Almogahed et al. [5], who emphasized the need for careful evaluation of refactoring techniques for their impact on security metrics.

Alshammari et al.[6] assessed the impact of general refactoring rules on the security of object-oriented applications. They evaluated security metrics before and after refactoring and identified which refactoring steps could enhance the security level of a program, particularly from the perspective of potential information flow. Similar to the trade-offs analysis by Abid et al. [1], Mumtaz et al. [101] studied the impact of general code refactoring on security by removing security-related code smells through refactoring, particularly by addressing feature envy with general move method techniques.

Ikegami et al. [72] looked into the integration of general refactoring techniques into security vulnerability fixes through a case study of Maven libraries. They reported that a significant proportion of vulnerability fixes involved refactoring actions, suggesting potential for automated tool support, which was explored by Maruyama and Omori [98] and Ghaith et al. [54].

Elkhail et al. [40] studied the relationship between code smells and security vulnerabilities, reporting that certain smells have a substantial impact on security, a perspective that complements the empirical findings of Codabux et al.'s work [70] which investigated the effects of specific refactorings such as Inline Method and Extract Interface on the security profile of applications.

Almogahed et al. [5] investigated the effect of commonly used refactoring techniques on the security attribute of information hiding through case studies. Their multi-case analysis categorizes refactoring techniques based on security metrics. The study discussed the importance of not indiscriminately applying refactoring techniques, advising developers to thoughtfully assess the advantages and disadvantages of each.

While prior studies have explored the impact of general refactoring techniques on software security, they often lack a focus on Java-based systems and do not provide concrete refactoring solutions to fix specific vulnerabilities. In contrast, this work differs by emphasizing Java-specific vulnerabilities and supporting broad vulnerability coverage through structured refactoring techniques, which are implemented and integrated within VulnFixAI.

### 2.0.2 Traditional techniques

Traditional techniques for vulnerability detection and fixing primarily rely on static and dynamic program analysis, often incorporating symbolic execution[120]. Ma et al. [94] introduced CDRep, a tool for automatically repairing cryptographic misuse

vulnerabilities in Android apps using predefined fix patterns, demonstrating the feasibility of automated patching for security-specific defects.

Hong et al. [64] presented SAVER, a scalable and precise system for automatically repairing memory errors, combining static analysis with dynamic validation to ensure both safety and correctness of generated patches. Van Tonder and Le Goues [147]developed a static automated program repair approach targeting heap property violations, using heap analysis and synthesis techniques to generate safe and correct fixes without requiring program execution.

Zhang et al. [162] proposed a vulnerability repair approach based on inductive inference, which learns generalizable patch patterns from examples and applies them to fix similar vulnerabilities across programs While these methods have been instrumental in detecting security flaws, they suffer from false positives, high computational costs, and limited generalizability(e.g., [95, 126, 167]).

VulnFixAI differs from traditional approaches by combining fine-tuned DSLLMs with structured refactoring algorithms. Unlike tools like CDRep and SAVER [64, 94], which focus on domain-specific fixes, VulnFixAI generalizes across multiple CWE types. It is trained on a large dataset of real-world Java vulnerabilities, enabling it to generate security-aware, CWE-aligned fixes. This hybrid approach ensures high-precision detection and scalable, consistent remediation with fewer false positives and incorrect fixes.

### 2.0.3 Machine Learning And Deep Learning Techniques

Machine learning (ML) and deep learning (DL) techniques have emerged as promising solutions for automating both vulnerability detection and repair. Aivatoglou et al.[4] investigated the effectiveness of tree-based machine learning techniques for automatically classifying software vulnerabilities. Their approach leveraged textual descriptions from the National Vulnerability Database (NVD), achieving an accuracy of nearly 80%.

Similarly, Chen et al. [26] proposed VRepair, a neural model using transfer learning to repair vulnerabilities in C code. The model is pre-trained on a general bug fix corpus and fine-tuned on a smaller vulnerability dataset, showing strong performance over traditional models.

Atiiq et al.[11] explored CWE-specific vulnerability detection, demonstrating that training classifiers for individual CWE categories improves detection accuracy. Their findings highlighted the challenges of handling diverse vulnerability types using a single classifier. Fu et al. [50] developed a novel approach inspired by Vision Transformers that uses ”vulnerability queries” and attention mechanisms to guide models toward repairing vulnerable code sections more accurately. Yang et al. [159] proposed DLAP, a deep learning-augmented prompting framework that combines DL and LLM strengths to improve detection accuracy and interpretability. Wang et al. [152]introduced a Graph Confident Learning (GCL4SVD) model that enhances detection by capturing syntactic structure from code and cleaning label noise, outperforming state-of-the-art baselines.

Zhou et al. [165] proposed VulMaster, a Transformer-based model that improves automatic vulnerability repair by handling long code sequences, incorporating code structure, and leveraging CWE knowledge. It outperformed prior methods on a large C/C++ dataset in EM, BLEU, and CodeBLEU metrics. Jormakka [76] focused on leveraging natural language processing (NLP) techniques for classifying software vulnerabilities, emphasizing the importance of structured representations such as CWE and the Common Vulnerability Scoring System (CVSS). Aghaei and Al-Shaer [3] further proposed an automated neural network-based approach for vulnerability classification and mitigation, reinforcing the role of ML in improving software security.

Fu et al. [52]fine-tuned a T5 transformer model to generate security patches. They show the effectiveness of pre-trained models in learning context-aware vulnerability fixes. Similarly, Zhang et al. [160] investigating various pre-trained models for automated soft-

ware vulnerability repair assesses their effectiveness and limitations, highlighting the gap between academic benchmarks and real-world applications.

Fu et al. [51] introduced AIBugHunter, an AI-driven tool for vulnerability detection, classification, and repair, They find that AIBugHunter effectively categorizes different types of vulnerabilities and suggests relevant fixes, improving developer efficiency and reducing the manual burden of vulnerability management.

Pinconschi et al. [121] investigated the effectiveness of 10 state-of-the-art program repair tools on real-world C/C++ vulnerabilities, showing that only 54.6% of vulnerabilities could be fixed and highlighting limitations in generalization to security bugs.

Wu et al. [157] conducted a foundational investigation into LLM-based automated program repair (APR) for Java vulnerabilities, evaluating five LLMs (Codex, CodeGen, CodeT5, PLBART, InCoder) on Vul4J (25 CWEs) and VJBench (23 CWEs, including 12 new CWEs). The study emphasizes that neural models require extensive fine-tuning and domain-specific adaptations to perform well in real-world scenarios.

Berabi et al. [16] proposed DeepCode AI Fix, a system that applies targeted fixes using prompt-tuned LLMs. Their approach relied on security-focused fine-tuning and transformer-based text-to-code generation, achieving the removal of over 80% of reported vulnerabilities. Lin et al. [86] investigated binary-level vulnerability repair, proposing TemVUR, a template-based repair method for Java binaries. Their approach outperformed source-level methods on the Vul4J dataset, repairing 66.7% more vulnerabilities.

Nguyen et al. [105] investigated the detection of vulnerability-fixing commits at multiple code granularities, proposing a multi-granularity detector that integrates file, function, and line-level representations using DL. Their approach leveraged both syntactic and semantic features to improve detection accuracy across real-world repositories. In large-scale experiments on five datasets, the model achieved an F1 score of 84.6%, out-

performing state-of-the-art baselines by up to 17.2%, particularly excelling at identifying subtle fix patterns across granular levels.

Nguyen et al. [104] explored semantic-based program repair, proposing SemFix, which uses symbolic execution and SMT solving to generate repairs consistent with specification behavior. Their method showed higher correctness than syntactic fix tools. Similarly, Hua et al. [66] introduce SketchFix, a tool that utilizes on-demand candidate generation to reduce the overhead associated with traditional generate-and-validate approaches. Their method translates faulty programs into ”sketches” with placeholders, which are then filled during test execution, effectively pruning the search space. Evaluated on the Defects4J benchmark, SketchFix successfully repaired 19 out of 357 bugs, requiring only 1.6% of recompilations and 3.0% of executions compared to exhaustive methods.

Peng et al. [120] proposed ANGLE, a novel method that combines hierarchical graph refinement and context-aware representation learning. Their approach filtered noisy information and jointly leveraged Graph Transformers and GNNs to capture both global and local code dependencies. In evaluations across three benchmark datasets, ANGLE outperformed state-of-the-art models with accuracy improvements ranging from 34.27% to 161.93%, particularly excelling on large code graphs. Li et al. [84] propose CleanVul, a novel approach that leverages large language model (LLM) heuristics to filter and classify vulnerable code. Their method combined static heuristics with GPT model predictions to refine noisy labels in real-world code commits. On evaluation benchmarks, CleanVul improved precision by 12.5% and F1 score by 9.3% over traditional heuristic-only baselines, demonstrating its effectiveness in practical vulnerability detection tasks. Zhou et al. [163] proposed Colefunda, an explainable framework for detecting such covert fixes. Their method applied contrastive learning and feature disentanglement

to differentiate security-relevant changes from non-security-related ones in commit histories. Achieved an F1 score of 82.1%, outperforming prior tools by up to 15.3%.

Islam and Najafirad [73] investigated the use of reinforcement learning (RL) combined with LLMs for automated vulnerability repair, proposing a novel RL-guided framework that rewards secure and compilable code changes. Their method focused on repairing vulnerabilities such as injection flaws, buffer overflows, and improper access control. The system integrated LLM-generated candidates with a reward signal based on compilation success and security patch similarity. On real-world C/C++ vulnerability datasets, their approach achieved a repair success rate of 76.3%, outperforming baseline LLM patching by 19.8%.

Nong et al. [108] investigated the use of chain-of-thought (CoT) prompting with LLMs to improve the detection and repair of software vulnerabilities, proposing a novel framework that guides LLMs through intermediate reasoning steps before suggesting fixes. The model achieved a vulnerability fix accuracy of 81.2%, outperforming direct prompting baselines by 16.5%, and demonstrated improved interpretability in fix generation.

These studies highlight the importance of ML, and DL vulnerability detection and repair, noting that most existing solutions focus on detection and applying patching rather than structured, long-term remediation strategies. This work addresses these gaps by tailored refactoring solutions for software vulnerabilities.

Recent research has explored the potential of LLMs for automated vulnerability detection and repair. Wu et al. [156] compared several pre-trained models, including Codex and CodeT5, demonstrating that fine-tuning significantly improves their ability to generate vulnerability fixes. However, they identified dataset limitations as a major constraint affecting the reliability of LLM-generated fixes. Dozono et al. [36] further evaluated LLM-based security assessments across multiple programming languages,

emphasizing the need for high-quality datasets. Their study revealed that domain-specific fine-tuning of LLMs enhances vulnerability detection accuracy.

Unlike general-purpose LLMs, This work leverages a high-quality, CWE-labeled dataset focused on Java vulnerabilities. This domain-specific fine-tuning enables VulnFixAI to generate accurate, context-aware repairs aligned with real-world security requirements.

Despite the contributions of the existing work, there is a gap in the literature concerning Java-specific security research and tailored refactoring solutions for software vulnerabilities, which is addressed in this work through the development of VulnFixAI, an automated tool that integrates refactoring techniques.

Table 2.1summarizes the related work, including their approaches, tool support, targeted programming languages, fix focus, strengths, and limitations. This comparative overview highlights the gap in current solutions, particularly the lack of structured refactoring support for Java-specific vulnerabilities and the limited coverage of diverse CWE types, thereby motivating the need for this study's hybrid approach integrating DSLLMs with refactoring techniques.

Table 2.1
Summary of Related Works

| Paper | Approach | Tools | Language | Fix Focus | Strengths | Limitations |
|---|---|---|---|---|---|---|
| Abid et al. (2020) | Multi-objective refactoring | No | General | Refactor impact on security metrics | Tested on 30 OSS projects, shows trade-offs between security and quality | Lacks fix automation, sometimes worsens security while optimizing quality |
| Almogahed et al. (2022) | Security-metric classification of refactorings | No | General | Information hiding via refactoring | Case-based analysis using security metrics | Not generalizable, lacks implementation guidance or tooling |
| Alshammari et al. (2010) | Security impact of refactorings using metrics | No | General | Information flow | Shows measurable metric improvements | No tooling, abstract evaluation without practical fix strategies |
| Mumtaz et al. (2017) | Feature envy refactoring for security | No | General | Code smells and security | Measurable security gains through refactoring | Limited to selected smells like feature envy |
| Ikegami et al. (2022) | Study of refactoring in vulnerability fixes | No | Java | Refactoring patterns in real-world fixes | Found frequent use of refactoring in actual security patches | Lacks automation, observational only |
| Maruyama & Omori (2011) | Security-aware refactoring alerts | Yes | Java | Secure refactoring | Practical Java tooling that warns on refactoring impact | Only handles simple refactorings, not broad fix coverage |
| Elkhail et al. (2019) | Code smell–vulnerability correlation | No | General | Security implications of smells | Empirical link between smells and vulnerabilities | No fix proposals; correlation only |
| Codabux et al. (2023) | Impact of individual refactorings | No | General | Vulnerability probability from refactorings | Identifies risky refactoring types | Not CWE-specific, lacks automated tooling |
| Ma et al. (2016) | CDRep: Pattern-based crypto repair | Yes | Java | Cryptographic misuse | 94.5% repair success on crypto bugs | Manual fix patterns |
| Hong et al. (2020) | SAVER: Static+dynamic repair validation | Yes | C/C++ | Memory error repair | Ensures semantic and safe patches via tests/symbolic exec | Only for memory bugs; computationally expensive |
| Van Tonder & Le Goues (2018) | Static heap patch synthesis | No | C | Heap corruption repair | Patch synthesis without runtime validation | Limited to heap domain, lacks wider bug coverage |
| Zhang et al. (2022) | Inductive patch inference | No | General | Learning fix patterns from patches | Transferable patch logic via inference | Sensitive to training quality, risk of false positives |
| Aivatoglou et al. (2021) | Vulnerability classification from NVD | No | General | Text-based classification | 80% accuracy in NVD classification | Not code-aware, relies only on descriptions |
| Chen et al. (2022) | VRepair: Transfer learning for bug fixes | Yes | C | Neural repair model | Leverages pretrained bug fix corpora | Language-specific C, domain adaptation challenges |
| Atiiq et al. (2024) | CWE-specific classifier ensemble | No | General | Vulnerability detection | Improves precision with CWE alignment | Hard to scale for many CWEs, requires large labeled data |
| Fu et al. (2024) | Vision Transformer–based repair | No | General | Contextualized vulnerability repair | Attention model improves localization accuracy | High complexity, limited transparency |
| Wu et al. (2023) | LLMs on Java vulnerability datasets | No | Java | Vul4J/VJBench multi-CWE fix | Realistic Java evaluation, new benchmarks | Fix rate ¡25%, poor compilation |
| Liu et al. (2024) | Prompt engineering for LLM repair | Yes | General | Prompt-based fix generation | Outperforms baseline NMT repair models | Highly sensitive to prompt design, lacks CWE types |
| Zhou et al. (2024) | VulMaster Transformer | No | C/C++ | Long-function vulnerability repair | Uses code structure + expert knowledge | C/C++-only, expensive model,no Java support |
| Islam & Najafirad (2024) | RL + LLM (SecRepair) | No | C/C++ | Secure repair using semantic reward | 12% improvement with RL tuning | Complex tuning; instruction-heavy, C/C++-focused |
| Nong et al. (2024) | CoT prompting for vuln. repair | No | General | Identification, discovery, patching | 30.8%+ improvement in patch F1 | CoT slows generation, struggles with complex bugs |
| **This Work** | **Hybrid refactoring and DSLLM-based** | **Yes** | **Java** | **Multi-CWE fixes** | **Domain-specific refactoring, fine-tuned model, 89% effectiveness, broad CWE coverage** | **Focused on Java only** |

# CHAPTER THREE

# APPROACH

This chapter details the proposed approach. It begins by introduces a set of refactoring techniques aimed at mitigating various types of vulnerabilities. Building upon these strategies, VulnFixAI, an automated tool for vulnerability detection and repair, is presented.

### 3.0.1 Refactoring Techniques (Vulnerability Analysis)

In order to develop effective refactoring techniques, it is necessary to examine various samples of vulnerable code, categorize them based on their root causes, and identify their defining traits. The CWE system is utilized, as it provides a standardized taxonomy for classifying software weaknesses. These weaknesses represent potential issues in artifacts (e.g., design, code) that, if exploited, can lead to vulnerabilities impacting confidentiality, integrity, or availability. The CWE system categorizes these issues into Buffer Errors, Resource Management Errors, Improper Error Handling, Injection, Improper Access Control, Cross-Site Scripting, Sensitive Data Exposure, and Cryptographic Issues. Each category includes a set of CWE IDs (types), representing various vulnerability types. The CWE system provides a description for each CWE type along with a few examples mostly in C.

To better understand the categories, root causes, and common traits of CWE types in Java which this study focuses on, more samples were needed. However, there are few publicly available datasets for vulnerability in Java that contain the code snippets.

To address this, CWE knowledge from the official CWE database [29] is utilized, focusing on CWE ID, CWE names, vulnerability descriptions, and code examples. Using

this knowledge as a foundation, ChatGPT-4 is employed to generate Java samples for each CWE type, ChatGPT-4 was selected due to its superior natural language processing capabilities, which allow it to generate contextually accurate Java code snippets and detailed explanations, ensuring the validity and precision of the vulnerability samples needed for the study. The effectiveness of AI models such as ChatGPT and Copilot [56] in generating high-quality code snippets has been well-demonstrated [89, 33, 92, 91, 90]. Five Java samples are generated for each CWE type, resulting in a total dataset of 405 samples for 81 CWE types. The dataset is made publicly available online [109].

The choice of five examples per CWE ID was made based on exploratory sampling using ChatGPT 4. It was observed that after generating approximately five samples for a given CWE type, the model began to produce examples with repetitive structures, similar logic, and limited variability. As a result, additional samples offered diminishing returns in terms of exposing new patterns or insights. Therefore, five samples per CWE ID were selected as a practical threshold that balances representational diversity with generation quality. This allowed us to capture varied manifestations of each vulnerability type without incurring redundancy in the dataset.

Initially, 400 unique CWE IDs were identified from the official CWE database [29]. However, only 81 CWE types were selected and categorized into eight groups. This subset was chosen because these CWE types exhibit clear semantic and structural similarities, enabling the meaningful development of targeted refactoring techniques. The remaining CWEs were excluded primarily because they lacked sufficient contextual alignment or were too heterogeneous to be grouped effectively under a common remediation strategy.

Each generated sample was thoroughly evaluated manually to ensure its validity against the description of the corresponding CWE type. The performance of ChatGPT in generating vulnerable code examples based on CWE is discussed in Section 3.0.1.1.

The below is the prompt template that was used for generation.

> *"Provide 5 different Java examples for the vulnerability type* **{`CWE ID, CWE Name`}**. *I will also give you the official CWE description* **{`vulnerability descriptions`}** *and corresponding code examples in other programming languages* {`code examples`} *to help guide the generation of the Java examples."*

The CWE ID, CWE Name, vulnerability descriptions, and code examples in the prompt are dynamically filled with the actual CWE identifier, name, official vulnerability description, and corresponding code examples (typically in C or other languages) as provided on the official CWE webpage for each vulnerability.

Analysis of the generated samples indicates that the root causes associated with CWE types within the same CWE category vary significantly, displaying distinct traits that challenge the development of a refactoring solution for the specific category. While the CWE system broadly categorizes software vulnerabilities, such classifications often hide the underlying causes and shared remediation patterns across CWE types. For instance, CWE-89 (SQL Injection) and CWE-77 (Command Injection) both differ in terms of their execution context and attack surface targeting databases and operating system commands, respectively. Despite these differences, they share the same root cause, unsanitized user input that is improperly embedded into an execution context (e.g., SQL queries, shell commands).

To address this issue, a new vulnerability taxonomy is introduced that groups CWE types based on their root causes and common behavioral characteristics. This refined classification presented in Table 3.1, supports the development of category-specific refactoring techniques tailored to the practical needs of mitigating vulnerabilities in Java systems.

- *Lack of Boundary Checks (LOBC)*: This category includes vulnerabilities caused by the absence of boundary checks within memory buffers. Such omissions can

lead to buffer overflows, a prevalent security issue where data exceeds the memory buffer's boundary, potentially overwriting adjacent memory. These vulnerabilities are generally considered high severity due to their potential to allow attackers to execute malicious code, leading to data corruption or crashes.

- *Improper Array Copying (IAC)*: Vulnerabilities in this category arise from incorrect array copying operations. These might include improper handling of array size, leading to buffer overflows, memory leaks, and out-of-bound reads. These vulnerabilities are generally considered high severity, as they compromise memory safety, leading to potential unauthorized access to sensitive data or system instability.

- *Lack of Input Sanitization (LOIS)*: This category deals with vulnerabilities stemming from inadequate sanitization of user inputs. Without proper checks, inputs may trigger SQL injection, XSS, command injection, or XML injection. These vulnerabilities are typically considered high severity, given that exploitation can lead to unauthorized data access, system control, or other malicious activities.

- *Improper Numeric Handling (INH)*: This category includes vulnerabilities that are related to errors in numeric calculations, such as failing to handle integer overflows or underflows, which can cause crashes or unintended behaviors. The severity of these vulnerabilities can vary but often trends towards medium to high depending on the context and potential impact on the system's stability or data integrity.

- *Lack of Whitelist Validation (LOWV)*: This category involves vulnerabilities that arise from failing to validate input against a predefined whitelist, potentially leading to unauthorized access or code injection. They are typically considered high, especially if the vulnerability allows bypassing security mechanisms and executing malicious code.

- *Improper Type Handling (ITH)*: This category includes vulnerabilities that occur due to errors in handling data types, potentially causing type confusion, incorrect calculations, and risks like integer overflow. These vulnerabilities vary in severity from medium to high, often depending on the potential for exploitation to cause more severe secondary effects like unauthorized access or information disclosure.

- *Insecure Error Feedback (IER)*: This category includes vulnerabilities that arise from providing overly detailed error messages, which could inadvertently expose sensitive system information to attackers. They are generally considered medium severity because they do not directly lead to a breach but can significantly aid an attacker in exploiting other vulnerabilities.

- *Inadequate TrustChain Verification (ITV)*: This category pertains to vulnerabilities caused by insufficient verification of trust chains, including the validation of digital signatures and certificates, which may compromise data integrity and permit unauthorized code execution. These vulnerabilities are generally considered high severity due to the direct risk of executing malicious code or accessing unauthorized data.

It should be noted that the categorization of vulnerabilities does not always involve a clear delineation between categories, a single piece of vulnerable code can be interpreted as belonging to multiple categories, depending on the perspective from which they are analyzed. For example, CWE-120 (Classic Buffer Overflow) can be viewed under LOBC, IAC, and LOWV due to its implications in different operational contexts. Under LOBC, CWE-120 is concerned with the absence of boundary checks that prevent data from exceeding the buffer's allocated space, crucial for avoiding security breaches. In the context of IAC, it highlights the need for validating array sizes during data copying to ensure the destination array's capacity is not surpassed, which helps prevent buffer over-

Table 3.1
Vulnerability Types

| Vulnerability Category | CWE IDs | Severity |
|---|---|---|
| Lack of Boundary Checks (LOBC) | CWE-22, CWE-74, CWE-119,CWE-122, CWE-125, CWE-170, CWE-189, CWE-190, CWE-191, CWE-400, CWE-404, CWE-787, CWE-823, CWE-824, CWE-825 | High |
| Improper Array Copying (IAC) | CWE-120, CWE-121, CWE-122, CWE-124, CWE-126, CWE-129 | High |
| Lack of Input Sanitization (LOIS) | CWE-74, CWE-75, CWE-76,CWE-77, CWE-78, CWE-79, CWE-80, CWE-83, CWE-85, CWE-86, CWE-88, CWE-89, CWE-90, CWE-91, CWE-93, CWE-94, CWE-95, CWE-96, CWE-97,CWE-98, CWE-99, CWE-113, CWE-116, CWE-117, CWE-134, CWE-138, CWE-140, CWE-150, CWE-151, CWE-152, CWE-157, CWE-158, CWE-159, CWE-160, CWE-165, CWE-166, CWE-167, CWE-176,CWE-352, CWE-434, CWE-470, CWE-471, CWE-502, CWE-564, CWE-601, CWE-614, CWE-642, CWE-643, CWE-917 | High |
| Improper Numeric Handling (INH) | CWE-189, CWE-190, CWE-191, CWE-192, CWE-193, CWE-194, CWE-195, CWE-196, CWE-197, CWE-198, CWE-369, CWE-681, CWE-682, CWE-682, CWE-707 , CWE-770 | Medium/ High |
| Lack of Whitelist Validation (LOWV) | CWE-20, CWE-23, CWE-36, CWE-41, CWE-59, CWE-74, CWE-77, CWE-119, CWE-120, CWE-130, CWE-131, CWE-134,CWE-179, CWE-180, CWE-181, CWE-182, CWE-183, CWE-184, CWE-185, CWE-186, CWE-208, CWE-306,CWE-354, CWE-434, CWE-444, CWE-470, CWE-501, CWE-601, CWE-611, CWE-641 CWE-707, CWE-787, CWE-943, CWE-1173, CWE-1919 | High |
| Improper Type Handling (ITH) | CWE-20,CWE-173, CWE-190, CWE-191, CWE-415, CWE-416, CWE-476, CWE-486, CWE-580, CWE-588, CWE-617, CWE-664, CWE-665, CWE-674, CWE-681, CWE-682, CWE-697, CWE-704, CWE-763, CWE-824, CWE-835, CWE-843, CWE-909, CWE-1025, CWE-1330 | Medium/ High |
| Insecure Error Feedback (IER) | CWE-89, CWE-119, CWE-200, CWE-204, CWE-209,CWE-210, CWE-211, CWE-212, CWE-213, CWE-214, CWE-215, CWE-255, CWE-264, CWE-269, CWE-281, CWE-284, CWE-285, CWE-287, CWE-290, CWE-297, CWE-378,CWE-379, CWE-536, CWE-537, CWE-544, CWE-550, CWE-552, CWE-693, CWE-756, CWE-778, CWE-798, CWE-862 | Medium |
| Inadequate TrustChain Verification (ITV) | CWE-295,CWE-296, CWE-275, CWE-297, CWE-298, CWE-299, CWE-599, CWE-319,CWE-320, CWE-321, CWE-326, CWE-327,CWE-330, CWE-329, CWE-353, CWE-345, CWE-346, CWE-347, CWE-358,CWE-775, CWE-401,CWE-494, CWE-502, CWE-522, CWE-532, CWE-547, CWE-668,CWE-732, CWE-916, CWE-918, CWE-940, CWE-941 CWE-1021 | High |

flow vulnerabilities. The presence in LOWV can be justified by the need for validating all input data against expected sizes and formats before processing to prevent overflows. This suggests that software vulnerabilities may be multi-faceted, requiring distinct solutions tailored to address the same issue from different operational perspectives.

#### 3.0.1.1 Effectiveness of ChatGPT

The effectiveness of ChatGPT in generating vulnerable code examples extends beyond code snippet generation [81, 102]. it explicitly provides detailed explanations of vulnerabilities, pinpoints the exact locations of vulnerable code, and how to address identified vulnerabilities. These characteristics are essential for comprehensive vulnerability analysis tasks.

To validate the quality of the generated examples, a manual evaluation involving domain experts was conducted. Specifically, the first and second authors are both researchers with extensive experience in software security and vulnerability analysis. The validation process began with a careful review of the relevant CWE entries on the official CWE website [29], including detailed vulnerability descriptions and expert analyses. This provided a solid foundation for assessing the correctness of the examples.

Following this, the experts examined each Java code example generated by ChatGPT. Examples that accurately reflected the characteristics and behaviors of the specified CWE were labeled as "vulnerable," whereas those that failed to meet these expectations were marked as "non-vulnerable."

In cases where the annotators disagreed, a dedicated review meeting was held to discuss and resolve the discrepancies. Only examples that achieved consensus as "vulnerable" were included in the final dataset [109].

The manual investigation revealed that ChatGPT achieved an accuracy of 77.12% (455/590) in generating Java vulnerability examples based on CWE definitions. This high accuracy supports the explanation to some extent.

#### 3.0.1.2 Refactoring Techniques

This work develops refactoring techniques for each category listed in Table 3.1 by thoroughly analyzing the root causes and common traits of the samples of vulnerability

types within each category. This detailed analysis led to the development of eight distinct refactoring techniques, which are outlined in Table 3.2.

Table 3.2
Vulnerability Refactoring Techniques

| Refactoring Technique | Description | VC |
|---|---|---|
| Boundary Sentinel Refactoring (BSR) | Implements additional boundary checks in memory buffers to ensure all memory accesses remain within defined limits, preventing overflows and enhancing security. | LOBC |
| Safe Array Copy Refactoring (SACR) | Utilizes Java's built-in methods for array copying, such as System.arraycopy and Arrays.copyOf, which include inherent boundary checks to ensure data is copied safely and accurately. | IAC |
| Output Safety Refactoring (OSR) | Employs sanitization functions from libraries like OWASP Java Encoder, Apache Commons Text, or Jsoup to cleanse user inputs, thus neutralizing potentially harmful characters and preventing code injection vulnerabilities. | LOIS |
| Numeric Safety Refactoring (NSR) | Incorporates checks to validate data type sizes in arithmetic operations to prevent numeric overflows and underflows. | INH |
| Whitelist Validation Refactoring (WVR) | Enforces data validation against a defined whitelist, including range checks for numerical data and format validations for strings to ensure only permissible data is processed. | LOWV |
| Type Enforcement Refactoring (TER) | Implements stringent type checks and uses explicit parsing methods to maintain data type integrity and avoid type confusion errors, such as ensuring parsing operations like Integer.parseInt() are used to validate and convert data types accurately. | ITH |
| Input Feedback Refactoring (IFR) | Modifies error feedback to provide specific guidance without exposing sensitive system information, using custom error messages and controlled exception handling mechanisms to reduce information leakage risks. | IEF |
| TrustChain Verification Refactoring (TCVR) | Strengthens the verification processes for external inputs, including the validation of digital signatures and checksums, and the implementation of secure deserialization practices to protect against unauthorized data manipulation. | ITV |

VC: Vulnerability Category.

It is important to note that while traditional software refactoring is defined as modifying a program's internal structure without changing it is external behavior [48], the techniques proposed in this work extend beyond this strict interpretation. Some of the presented techniques (e.g., SACR, TER) align with classical refactoring by enhancing internal code structure and safety without modifying program output. However, other techniques (e.g., BSR, OSR, NSR, WVR, TCVR) may alter certain aspects of the program's runtime behavior to eliminate vulnerabilities or prevent security breaches. Therefore,

this work adopts a broader notion of security-oriented refactoring, encompassing both structural improvements and corrective transformations that aim to enhance code security while promoting maintainability and clarity.

#### 3.0.1.3 Boundary Sentinel Refactoring

The Boundary Sentinel Refactoring (BSR) technique specifically targets LOBC vulnerabilities concerning memory safety issues that arise when operations extend beyond the allocated memory buffer boundaries. This can manifest in several severe issues such as data corruption of adjacent memory segments, program crashes by segmentation faults, and security vulnerabilities by malicious code injection. The BSR method mitigates these risks by implementing additional checks – referred to as sentinels – at the boundaries of memory buffers. These sentinels verify that all memory access attempts are within the defined safe limits of the buffer, thereby preventing the aforementioned issues and enhancing the security and stability of the software.

Algorithm 1 describes the BSR technique. For each identified location in the code where buffers are accessed, it inserts a boundary check along with exception handling at that point. The $build\_check$ function generates a boundary check to ensure that any index or pointer used does not exceed the buffer's length; $index\_usage$ represents the index used to access elements within the buffer, and $buffer\_name$ represents the name of the buffer being accessed. The $build\_exception$ function generates exception handling for the boundary check. The $build\_new\_exp$ function constructs a new expression that incorporates the check with the exception handler before the original expression. This results in a statement like if ($index\_usage$ >= size of $buffer\_name$) then throw new ArrayIndexOutOfBoundsException("Buffer Overflow Prevented"), where the >= operator is crucial as it ensures that the index does not exceed the buffer size. Other operators (e.g., <, <=, >) are unsuitable for this purpose as they do not comprehensively prevent overflows in all

scenarios. If an out-of-bound access attempt is detected, a specific exception is thrown to prevent a buffer overflow, thereby enhancing the application's security and stability.

**Algorithm 1:** Boundary Sentinel Refactoring (BSR)

**Input:** Codebase containing buffer access points

**Output:** Refactored codebase with boundary checks

**Function** `BSR()`**:**

  **if** *line contains buffer access* **then**

    $original\_exp \leftarrow$ get the content;

    $index\_usage \leftarrow$ extract index or pointer from buffer access;

    $buffer\_name \leftarrow$ extract buffer name from access line;

    $check\_statement \leftarrow$ `build_check`$(index\_usage, buffer\_name)$;

    $exception \leftarrow$ `build_exception`$(check\_statement)$;

    $new\_exp \leftarrow$ `build_new_exp`$(check\_statement, exception, original\_exp)$;

    replace $original\_exp$ with $new\_exp$;

  **end**

To demonstrate the effectiveness of the BSR technique, consider the LOBC vulnerability in Listing 3.1. In this example, the $dataCopy()$ method attempts to copy data from a source array to a smaller destination array without boundary checks. This oversight can lead to a buffer overflow if the source array exceeds the destination array in size.

```
1 public void dataCopy(char[] source) {
2   char[] destination = new char[10];
3   for (int i = 0; i < source.length; i++) {
4     destination[i] = source[i]; // <——— LOBC Vulnerability (CWE—120, CWE—125)
5   }}
```

Listing 3.1: Before BSR

Listing 3.2 shows the refactored code where a boundary check is integrated within the loop. This check ensures that the index $i$ does not surpass the destination array's length. If a boundary violation occurs, an $ArrayIndexOutOfBoundsException$ is thrown, effectively preventing a buffer overflow and enhancing memory safety.

```
1 public void dataCopy(char[] source) {
2   char[] destination = new char[10];
3   for (int i = 0; i < source.length; i++) {
4     if (i >= destination.length) // <—— Fix by BSR
5     throw new ArrayIndexOutOfBoundsException("Buffer Overflow Prevented");
6     destination[i] = source[i]; }}
```

Listing 3.2: After BSR

Note that the vulnerability in Listing 3.1 can also be interpreted from the IAC or LOWV perspective, each leading to a different refactoring solution. From the IAC perspective, the problem is seen as unsafe array copying due to a lack of bounds checking, suggesting a solution that involves implementing safe copying practices to ensure that data does not exceed array boundaries. From the LOWV perspective, the issue is viewed as a failure to validate input against a set of allowed values or sizes, indicating a need for input validation to ensure data conforms to expected constraints before it is processed.

#### 3.0.1.4 Safe Array Copy Refactoring

The Safe Array Copy Refactoring (SACR) technique tackles IAC vulnerabilities concerning array copying without prior verification of the source array's size against the destination array's capacity, a common cause of buffer overflows. This technique ensures safety by implementing checks to confirm that the length of the data being copied does not surpass the smallest array size involved in the operation.

Algorithm 2 illustrates this process. For each location where arrays are copied, it introduces a safety check to ensure that the length of the copied data does not exceed the minimum length of the source and destination arrays. The $determine\_max\_length$ function calculates the maximum safe length for copying based on the lengths of the source and destination arrays using built-in functions (e.g., $Math.min()$), and it utilizes copying functions (e.g., $System.arraycopy()$, $Arrays.copyOf()$) for the copying process itself. Then, the length is passed to the $build\_statement$ function, which constructs a new safe copy statement by combining the source array, destination array, and the safe length.

**Algorithm 2:** Safe Array Copy Refactoring (SACR)

**Input:** Codebase containing array manipulation points

**Output:** Refactored code with safe copy checks

```
Function SACR():
    if line involves array manipulation then
        source_array ← extract source array from the location;
        destination_array ← extract destination array from the location;
        max_safe_length ← determine_max_length(source_array, destination_array);
        new_copy_statement ← build_statement(source_array, destination_array,
          max_safe_length);
        replace original array manipulation line with new_copy_statement;
    end
```

Consider the code snippet in Listing 3.3 containing an IAC vulnerability where the $storeUserData()$ method directly copies data from a string to a character array. This method does not check if the string's length exceeds the array's capacity, which can lead to a buffer overflow if the string is longer than the array.

```
1 public void storeUserData(String userData) {
2   char[] dataArray = new char[50];
3   userData.getChars(0, userData.length(), dataArray, 0); // <——— IAC Vulnerability ( CWE—120, CWE—122)
4 }
```

Listing 3.3: Before SACR

To address this vulnerability, the SACR technique is implemented as illustrated in Listing 3.4. In this refactored version, the method calculates the length of data to be copied using the minimum value between the string's length and the array's capacity. This ensures that only a safe amount of data is transferred to the array, thereby preventing any possibility of buffer overflow. The use of built-in methods like $Math.min()$ and $getChars()$ facilitates safe array copying, enhancing the application's stability and security against buffer overflow vulnerabilities.

```
1 public void storeUserData(String userData) {
2   char[] dataArray = new char[50];
3   int lengthToCopy = Math.min(userData.length(), dataArray.length); // <—— Fix by SACR
4   userData.getChars(0, lengthToCopy, dataArray, 0);
5 }
```

Listing 3.4: After SACR

#### 3.0.1.5 Output Safety Refactoring

The Output Safety Refactoring (OSR) technique targets LOIS vulnerabilities arising from the incorrect handling of user input in dynamic web content generation, such as HTML or JavaScript. This type of content, often generated from user inputs (e.g., comments, profile updates), poses security risks if the inputs are not properly sanitized or encoded. To mitigate these risks, OSR employs established libraries such as OWASP Encoder for HTML and OWASP ESAPI for JavaScript, as well as URL encoding functions, to ensure that user input is safely rendered in the output.

Algorithm 3 outlines the OSR process. For each location where dynamic content is generated, it applies encoding to prevent XSS and other injection attacks by sanitizing the dynamic content before it is rendered on the web page. The *encode* function sanitizes the dynamic content, and its output is used by the $build\_output\_exp$ function to construct a sanitization statement, ensuring that the output data does not contain unsafe content.

**Algorithm 3:** Output Safety Refactoring (OSR)

**Input:** Codebase with points of dynamic content generation

**Output:** Refactored code for safe output handling

```
1 Function OSR():
2     if line includes dynamic content generation then
3         dynamic_content ← extract dynamic content;
4         encoded_content ← encode(dynamic_content);
5         new_output_exp ← build_output_line(encoded_content);
6         replace original expression with new_output_exp;
7     end
```

Listing 3.5 shows an example of an LOIS vulnerability where the web application incorrectly renders a user-provided username directly in an HTML greeting message without sanitization. This leaves the application vulnerable to XSS attacks as malicious scripts could be injected via the username parameter.

```
String username = request.getParameter("username");
String htmlOutput = "<p>Welcome, " + username + "</p>";
response.getWriter().println(htmlOutput); // <——— LOIS Vulnerability (CWE—79, CWE—116, CWE—80)
```

Listing 3.5: Before OSR

This security flaw can be addressed by the OSR technique as shown in Listing 3.6. In this refactored version, the username is sanitized using the OWASP Java Encoder before it is embedded into the HTML output. This approach prevents XSS by ensuring that any potentially harmful scripts in the user input are rendered harmless.

```
import org.owasp.encoder.Encode;
String username = request.getParameter("username");
String sanitizedUsername = Encode.forHtml(username);
String htmlOutput = "<p>Welcome, " + sanitizedUsername + "</p>"; // <—— Fix by OSR
response.getWriter().println(htmlOutput);
```

Listing 3.6: After OSR

#### 3.0.1.6 Numeric Safety Refactoring

The Numeric Safety Refactoring (NSR) technique addresses INH risks associated with arithmetic operations that might result in integer overflow or underflow, potentially causing erroneous outcomes and security vulnerabilities. NSR addresses these risks by inserting checks before arithmetic calculations to ensure values remain within the permissible limits of the data types used. Algorithm 4 details the NSR process. For each location of arithmetic operations, it performs conditional checks to preemptively catch overflow or underflow conditions. If an overflow or underflow is detected, the algorithm triggers an exception to prevent the operation from proceeding, thus maintaining the integrity of the application's operations. The $build\_check$ function constructs a limit check for the operands involved in the operation using the data type limits (e.g., $Integer.MAX\_VALUE$, $Integer.MIN\_VALUE$) of the operands. The $build\_exception$ function generates exception handling for the check. The $new\_operation\_exp$ function

then constructs a new operation expression that incorporates the limit check with exception handling before the original operation.

**Algorithm 4:** Numeric Safety Refactoring (NSR)

**Input:** Codebase with arithmetic operations

**Output:** Refactored code with overflow/underflow checks

```
Function NSR():
    if line includes arithmetic operations then
        operation_exp ← extract arithmetic operation;
        check_statement ← build_check(operation_exp);
        exception ← build_exception(check_statement);
        new_operation_exp ← build_safe_numeric_operation(check_statement, exception,
          operation_exp);
        replace original operation expression with new_operation_exp;
    end
```

Listing 3.7 shows an example of an INH vulnerability where the $add()$ method performs a straightforward addition of two integers. This method does not account for the possibility of integer overflow which occurs if the sum exceeds the storage capacity of an integer. This oversight can lead to erroneous results and potential security vulnerabilities.

```
1 public int add(int a, int b) {
2   return a + b; // <——— INH Vulnerability (CWE—190, CWE—191)
3 }
```

Listing 3.7: Before NSR

To address the vulnerability, the NSR technique is applied as shown in Listing 3.8. The refactored version includes an overflow check prior to the addition operation. The condition $Integer.MAX_VALUE - a < b$ is used instead of $a + b > Integer.MAX_VALUE$ to preemptively prevent the operation from resulting in a value exceeding the integer's limit, which could cause an overflow even during the check itself. If the condition evaluates as true, indicating a potential overflow, an $ArithmeticException$ is thrown.

```
1 public int add(int a, int b) {
2   if (Integer.MAX_VALUE — a < b) // <—— Fix by NSR
3   throw new ArithmeticException("Integer overflow");
4   return a + b;
5    }
```

Listing 3.8: After NSR

#### 3.0.1.7 Whitelist Validation Refactoring

The Whitelist Validation Refactoring (WVR) technique mitigates LOWV risks associated with accepting unchecked user inputs from forms, URL parameters, and other entry points, which can lead to injection attacks and data integrity issues. To address these vulnerabilities, WVR defines specific patterns or criteria whitelists that all input must conform to before being processed. Algorithm 5 describes the WVR process. For each location where user input is handled, the $input_data$ function identifies the input data, and the $pattern$ function determines a whitelist for handling the data. The $validation$ function constructs a validation check based on the whitelist pattern. The output from this function is then used by the $build_input_handling$ function to create new input handling mechanisms.

**Algorithm 5:** Whitelist Validation Refactoring (WVR)

**Input:** Codebase with user input handling

**Output:** Refactored code incorporating robust input validation

```
Function WVR():
    if line handles user input then
        input_data ← extract input data from original expression;
        pattern ← determine whitelist for input_data;
        validation ← build_validation(input_data, pattern);
        new_input_handling ← build_input_handling(validation);
        replace original input expression with new_input_handling;
    end
```

Listing 3.9 shows an example of an LOWV vulnerability. In the code, the $processFormData()$ method retrieves a username from an HTTP request without validating the input, which poses a risk of accepting potentially malicious or malformed data, leading to various injection attacks.

```
1 public void processFormData(HttpServletRequest request) {
2   String username = request.getParameter("username"); // <——— LOWV Vulnerability (CWE—20, CWE—707)
3 }
```

Listing 3.9: Before WVR

To address the vulnerability, WVR is applied as shown in Listing 3.10. In the refactored version, a whitelist validation checks the username against a predefined regular expression pattern that allows only alphanumeric characters, underscores, or hyphens, and restricts the username length to between 3 and 15 characters. If the input matches this pattern, it is processed; otherwise, the system triggers error handling mechanisms.

```
1 private static final String USERNAME_REGEX = "^[a-zA-Z0-9_-]{3,15}$";
2     public void processFormData(HttpServletRequest request) {
3   String username = request.getParameter("username");
4   if (username.matches(USERNAME_REGEX)) { // <-- Fix by WVR
5     processValidUsername(username);
6   } else {
7     handleInvalidUsername();
8   }
9 }
```

Listing 3.10: After WVR

#### 3.0.1.8 Type Enforcement Refactoring

The Type Enforcement Refactoring (TER) technique aims to prevent ITH vulnerabilities arising from loose type conversions or casting. Algorithm 6 describes the TER refactoring. For each location of type conversions, it enforces strict typing rules based on the expected type. The $build_exception$ function constructs an exception handler for the type, and the $build_strict_conversion$ function applies a strict conversion method (e.g., $Integer.parseInt()$ for the *integer* type) and produces a new conversion with exception handling that replaces the original one.

**Algorithm 6:** Type Enforcement Refactoring (TER)

**Input:** Codebase involving type conversions and data handling

**Output:** Refactored code that enforces strict type compliance

```
Function TER():
    if line involves type conversion (e.g., casting, parsing) then
        type_expected ← determine expected type for conversion;
        exception ← build_exception(type_expected);
        new_conversion ← build_strict_conversion(type_expected, exception);
        replace original conversion with new_conversion;
    end
```

To illustrate the effectiveness of the TER technique, consider the example in Listing 3.11. In the code, the $processUserId()$ method attempts a direct cast from a *String* to an *int*, which is not only incorrect in Java but also risky because it can lead to runtime exceptions like $NumberFormatException$. This approach can lead to type confusion errors, as Java does not support casting directly from a *String* to an *int*.

```
1 public void processUserId(String userIdString) {
2   int userId = (int) userIdString; // <——— ITH Vulnerability (CWE—683, CWE—686, CWE—688)
3 }
```

Listing 3.11: Before TER

To address the issue, TER is applied as shown in Listing 3.12. The refactored code replaces the unsafe casting with a safer parsing method, $Integer.parseInt()$, enclosed in a try-catch block for handling potential parsing errors.

```
1 public void processUserId(String userIdString) {
2   try {
3     int userId = Integer.parseInt(userIdString); // <—— Fix by TER
4     furtherProcessing(userId
5   } catch (NumberFormatException e) {
6     logError("Invalid user ID format: " + userIdString, e);
7   }
8 }
```

Listing 3.12: After TER

#### 3.0.1.9 Input Feedback Refactoring

The Input Feedback Refactoring (IFR) technique aims to resolve IEF vulnerabilities arising from user feedback or error messages that could inadvertently disclose sensitive system information, aiding potential attackers. IFR address such vulnerabilities by replacing error messages with feedback that does not expose critical system details or the application's underlying logic while being helpful for the user. Algorithm 7 describes the IFR technique. For each location of user feedback or error messages being generated, it evaluates the feedback for the presence of sensitive information such as file paths, system configurations, or database details. If such information is found, it replaces

the feedback with content that lacks exploitable details. The $build\_feedback$ function produces neutralized feedback devoid of sensitive information.

**Algorithm 7:** Input Feedback Refactoring (IFR)

**Input:** Codebase containing user feedback and error messages

**Output:** Refactored code to include improved and secure feedback

```
Function IFR():
    if line displays user feedback then
        original_feedback ← extract feedback;
        if use feedback contains sensitive information then
            neutralized_feedback ← build_feedback(original_feedback);
        end
        replace original feedback with neutralized_feedback;
    end
```

Listing 3.13 shows an example of an IFR vulnerability. It involves a database query operation within an authorization check. If an $SQLException$ occurs, it logs a detailed error message that includes the query itself. This could inadvertently expose sensitive information about the database structure, potentially aiding attackers in formulating targeted database attacks.

```
try


{
  if (isAuthorizedUser(username)) {
    query = "SELECT * FROM accounts
    WHERE owner = " + username + " AND accountID = " + accountNumber;
    DatabaseManager dbManager = new DatabaseManager();
    Connection conn = dbManager.getConnection();
    Statement stmt = conn.createStatement();
    ResultSet queryResult = stmt.executeQuery(query);
    userAccount = (BankAccount) queryResult.getObject(accountNumber);
  }
}
catch(
SQLException ex)


{
  String logMessage = "Unable to retrieve account information from database,\nquery: " + query; // <——— IEF Vulnerability CWE—209, CWE—215)
  Logger.getLogger(BankManager.class.getName()).log(Level.SEVERE, logMessage, ex);
}
```

Listing 3.13: Before IFR

To address the risk, the IFR technique is applied as shown in Listing 3.14. The refactored code modifies the error handling to generalize the feedback, preventing the error message from disclosing the SQL query.

```
1 catch(
2 SQLException ex)
3
4 {
5   String logMessage =
6   "Unable to retrieve account information from the database."; // <-- Fix by IFR
7   Logger.getLogger(BankManager.class.getName()).log(Level.SEVERE, logMessage, ex);
8 }
```

Listing 3.14: After IFR

#### 3.0.1.10 TrustChain Verification Refactoring

The TrustChain Verification Refactoring (TCVR) technique targets ITV vulnerabilities associated with executing code or deserializing data from untrusted sources, which can lead to security breaches such as the execution of malicious code or data integrity issues.

TCVR addresses such vunlerabilities by verifying external inputs before utilization using techniques such as checking digital signatures, employing checksums for data integrity, and secure deserialization practices (e.g., class whitelisting, custom deserialization methods [124]). Algorithm 8 describe the TCVR process. For each location involving interactions with external data or code (e.g., downloads, API calls, file reads), it constructs a verification based on the interaction details and proceeds with the interaction only if the verification passes. The $build_verified_execution$ function integrates this verification with the execution of interactions, ensuring the integrity and authenticity of the interaction details before execution.

Consider the example in Listing 3.15. The code downloads a file from the given URL and saves it as "downloaded_code.jar". It then transfers the data from the URL to the local file. After the download, the code is executed. However, the

**Algorithm 8:** TrustChain Verification Refactoring (TCVR)

**Input:** Codebase involving external data or code
**Output:** Refactored codebase with verification mechanisms

```
Function TCVR():
    for each line in codebase do
        if line involves external data or code then
            interaction_details ← extract content;
            verification ← construct verification for interaction_details;
            verified_execution ← build_verified_execution(verification);
            replace original execution with verified_execution;
        end
    end
```

*downloadAndExecute*() method lacks verification of the downloaded file, potentially exposing the system to malicious code execution or data integrity compromises.

```
1 public class CodeDownloader {
2   public void downloadAndExecute(String url) throws IOException {
3     URL website = new URL(url);
4     ReadableByteChannel rbc = Channels.newChannel(website.openStream());
5     FileOutputStream fos = new FileOutputStream("downloaded_code.jar");
6     fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
7     executeDownloadedCode("downloaded_code.jar"); // <——— ITV Vulnerability (CWE—494, CWE—807, CWE—915)
8   }
9 }
```

Listing 3.15: Before TCVR

To address the risk, the TCVR technique is applied as shown in Listing 3.16. In the refactored code, checksum verification is added before executing the downloaded file. This involves calculating a SHA-256 checksum of the downloaded file and comparing it with a pre-defined expected checksum to verify the file's integrity. If the checksums match, the code proceeds with the execution; otherwise, it logs a failure message and refrains from executing the potentially compromised file.

```
public void downloadAndExecute(String url)
throws IOException, NoSuchAlgorithmException {
  // Same code as in Listing 15
  if (verifyChecksum("downloaded_code.jar", "expected_checksum")) { // <—— Fix by TCVR
    executeDownloadedCode("downloaded_code.jar");
  } else {
    System.out.println("Downloaded code integrity check failed.");
  }
}

private boolean verifyChecksum(String filePath, String expectedChecksum)
throws NoSuchAlgorithmException, IOException {
  MessageDigest sha256 = MessageDigest.getInstance("SHA—256");
  try (InputStream fis = new FileInputStream(filePath)) {
    byte[] dataBytes = new byte[1024];
    int nread;
    while ((nread = fis.read(dataBytes)) != —1) {
      sha256.update(dataBytes, 0, nread);
    }
  }
  byte[] mdbytes = sha256.digest();
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < mdbytes.length; i++) {
    sb.append(Integer.toString((mdbytes[i] & 0xff) + 0x100, 16).substring(1));
  }
  return sb.toString().equals(expectedChecksum);
}
```

Listing 3.16: After TCVR

#### 3.0.1.11 Complex Refactoring Scenarios: A Case Study

While the initial examples focus on fundamental cases to illustrate the refactoring approach, the proposed techniques can be extended to handle more complex real-world scenarios. For instance, refactoring techniques involving memory buffer management must adapt to dynamically allocated memory buffers, where buffer sizes are determined at runtime rather than hardcoded. Instead of static checks (e.g., $if(i >= 10)$), dynamic bounds can be retrieved using methods like buffer.length. Similarly, in regular expression-based validation, manually defining patterns may not always be practical. A more scalable approach involves predefined rule-based patterns from security libraries (e.g., OWASP Java Encoder). This allows input validation mechanisms to be adaptable

rather than requiring hardcoded patterns. Furthermore, using fixed numbers, like hard-coded hash values, often called (magic numbers), can create security risks. The proposed approach recommends generating cryptographic signatures dynamically at build time, ensuring integrity without introducing static magic numbers.

To demonstrate the application of the proposed refactoring techniques to complex real-world scenarios, a case study is presented analyzing CVE-2021-44228 [125], which affected the Apache Log4j2 [8] library (versions 2.0-beta9 through 2.14.1). This vulnerability arises from input validation and trust verification failures in Log4j's JNDI lookup feature.

If a user-controlled input, such as an HTTP header, contains a specially crafted string like $jndi:ldap://attacker.com/exploit$, Log4j resolves it using JNDI, potentially fetching and executing remote code from an attacker-controlled server. The severity of this vulnerability was rated CVSS 10.0 (Critical).

Consider a web application that logs an HTTP header using Log4j2, as shown in Listing 3.17. In vulnerable versions, any $jndi:\ldots$ sequence in the input is interpreted by Log4j's lookup mechanism, initiating a JNDI request without sanitization or allowlisting.

```
1  String userAgent = request.getHeader("User—Agent");
2  // Example input: '${jndi:ldap://attacker.com/a}' will invoke a JNDI lookup.
3  logger.warn("Request User—Agent: {}", userAgent); // ——> LOIS & LOWV
```

Listing 3.17: Before WVR

To address this vulnerability, WVR and TER were applied, as illustrated in Listing 3.18. The refactored code introduces an allowedHosts list to restrict JNDI lookups to trusted endpoints. If the host is not explicitly permitted, the request is aborted. Additionally, an allowedClasses list ensures that only safe object types can be deserialized.

```
if("ldap".

equalsIgnoreCase(uri.getScheme())||"ldaps".

equalsIgnoreCase(uri.getScheme()))

{
  if (!allowedHosts.contains(uri.getHost())) { // ——> WVR
    LOGGER.warn("Attempt to access LDAP server not in allowed list");
    return null;
  }
  Attributes attrs = context.getAttributes(name); // ——> TER
  if (attrs != null) {
    String className = getJavaClassName(attrs);
    if (!allowedClasses.contains(className)) {
      LOGGER.warn("Deserialization of {} is not allowed", className);
      return null;
    }
  }
}
return context.lookup(name);
```

Listing 3.18: After WVR

The WVR technique enforces that only pre-approved endpoints can be accessed, mitigating LOWV. The TER technique ensures that only trusted types are deserialized, addressing ITH. These changes significantly reduce the attack surface by applying strict host and type validation.

This case required a combination of defenses. This further emphasizes the importance of proactively applying secure design principles, such as default-deny policies and strict input validation, during software development.

### 3.0.2 VulnFixAI

The proposed VulnFixAI tool for automated vulnerability detection and repair in Java code comprises three major components Data Acquisition, Data Preparation, and VulnFixAI Architecture. The detailed architecture and various subcomponents are shown in Figure 3.1. The main tasks in the Data Acquisition module include identifying candidate vulnerabilities using the static analysis tool CodeQL. The Data Preparation module

encompasses data cleaning to remove duplicates and noise, and applying systematic refactoring techniques to address vulnerabilities. The VulnFixAI Architecture includes formatting the dataset for model training, building, and fine-tuning the llama 3.2 (3B) model, integrating it into a full-stack application for vulnerability detection and repair.

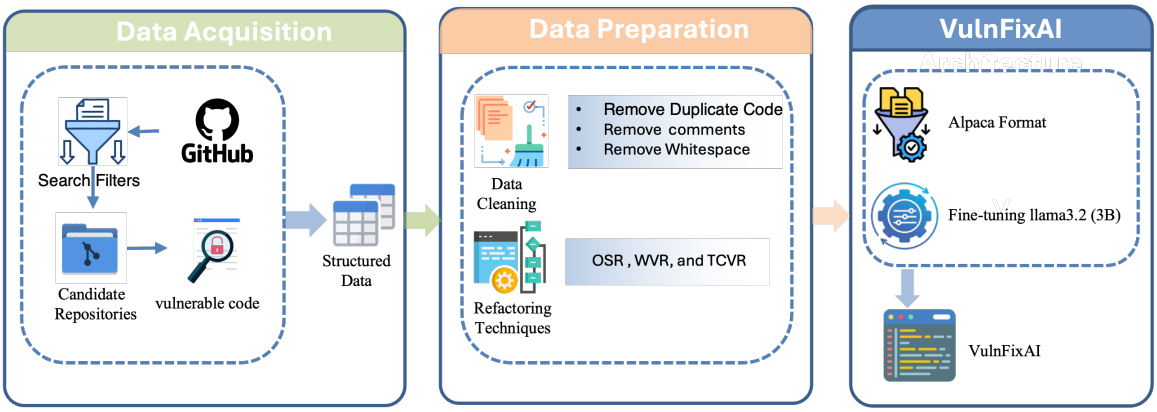



Figure 3.1: Overview of VulnFixAI

### 3.0.2.1 Data Acquisition

A total of 10,000 vulnerable Java code snippets were systematically gathered from 907 open-source GitHub projects. As shown in Table 3.3, the repositories were selected based on a multiple factors to ensure their relevance, diversity, and suitability for vulnerability detection. The selection was facilitated by utilizing GitHub's search filters to find projects with a high number of stars, and forks, as these indicators reflect the significance and continued development of the projects.

Each repository was cloned to a local environment where necessary dependencies, such as the Java JDK, were installed to facilitate a thorough and consistent analysis. The CodeQL static analysis tool [55] was utilized to scan the repositories for candidate

Table 3.3
Characteristics of Selected Repositories

| Attribute | Details |
|---|---|
| Number of Projects | 907 |
| Number of Stars | 500 to 77607 |
| Lines of Code (LOC) | 6,026 - 1,022,939,232 |
| Application Domains | Android, Web, Desktop Applications |
| Project Age Range | 2008 - 2024 |
| Project Size Range | 30 KB - 7.4 GB |

vulnerable code snippets. CodeQL was selected due to its well-documented efficacy in security analysis, as evidenced by its extensive validation in the scientific literature (e.g., [13, 85, 87, 135, 119]), making it a trusted tool for vulnerability detection in software systems.

#### 3.0.2.2 Data Preparation

Following extraction, a data cleaning process was conducted to remove duplicate code, unnecessary comments, and extra whitespace. To ensure data quality, a manual validation was performed on a randomly selected sample of 1000 class and function level snippets, verifying the vulnerabilities and minimizing false positives. The validation was conducted independently by two experienced security researchers, each reviewing snippets individually to determine the correctness of identified vulnerabilities. Discrepancies between reviewers were discussed and resolved jointly, ensuring consistency and minimizing subjective bias. This rigorous cross-verification approach confirmed the accuracy and reliability of the dataset for subsequent training and evaluation. =Finally, the refactoring techniques OSR, WVR, and TCVR were applied to fix the vulnerabilities. These techniques are described in detail in Section 3.0.1.2. The quality of the fixes was evaluated by running the CodeQL static analysis tool on the refactored code snippets. Each applied refactoring technique ensured that vulnerabilities were addressed. Table 3.4 shows

a statistical summary of the categorized vulnerabilities included in the dataset. Each vulnerability category is listed along with the total number of collected code snippets.

Table 3.4
Distribution of Vulnerability Categories in the Dataset

| Vulnerability Category | Total Code Snippets |
|---|---|
| Inadequate TrustChain Verification (ITV) | 2000 |
| Lack of Input Sanitization (LOIS) | 5000 |
| Lack of Whitelist Validation (LOWV) | 3000 |

#### 3.0.2.3 VulnFixAI Architecture

The rapid evolution of AI and ML has facilitated the development of LLMs such as GPT-4, Meta's Llama, and Google's Gemini. While LLMs have demonstrated remarkable capabilities in text understanding, and code completion, their deployment for software security tasks poses challenges, including high computational costs, and data privacy. To overcome these limitations, this study investigates the fine-tuning of DSLLMs for automated vulnerability detection and repair with a specific focus on Meta's Llama 3.2 (3B parameters).

Llama 3.2 (3B parameters) was selected due to its optimal balance between computational efficiency and performance. Compared to larger models (e.g. GPT-4), Llama 3.2 (3B) can be fine-tuned effectively on affordable hardware, facilitating practical local deployment and integration into continuous development workflows. Moreover, its smaller size enables efficient domain-specific fine-tuning, significantly improving accuracy and precision in both vulnerability detection and repair tasks without imposing substantial computational overhead.

| Instruction | Input | Output |
|---|---|---|
| Analyze the code for {CWE ID}. Identify the vulnerable line and explain why. | {Code Snippet} | {Vulnerable Line + Explanation} |
| Apply a fix for the {CWE ID} vulnerability in the code snippet. | {Code Snippet} | {Code Fix} |

Table 3.5
Alpaca format

#### 3.0.2.4 Alpaca Format

Fine-tuning requires structuring the dataset into an instruction-based format compatible with LLAMA 3.1. The Alpaca format is adopted due to its structured nature, which enhances supervised fine-tuning by providing clear, prompt-based interactions [161, 30].

As shown in Table 3.5, the Alpaca format provides a structured approach to organizing the dataset for fine-tuning by clearly defining instruction, input, and output fields. The instruction field specifies the task (e.g., vulnerability identification or code repair), the input field supplies the relevant data (e.g., a Java code snippet with a vulnerability), and the output field provides the expected result (e.g., vulnerability explanation or corrected code). For this dataset, two primary instructions are defined:

- Identifying Vulnerabilities: The model analyzes a Java code snippet to pinpoint the exact line containing a vulnerability tied to a specific CWE ID with an explanation of the flaw.
- Applying Fixes: The model applies a security fix to a vulnerable code snippet based on a specified CWE ID, producing corrected code.

This format enables accurate, context-aware learning, enhancing the model's ability to detect and remediate vulnerabilities effectively.

| Parameter | Details |
|---|---|
| Base Model | Llama 3.2-3B |
| Training Framework | Unsloth library for optimized parameter management |
| Optimization | AdamW optimizer with 8-bit precision and a linear learning rate scheduler |
| Training Steps | 60 iterations with a batch size of 8. |
| Memory Utilization | 4.363 GB. |

Table 3.6
Model Parameters

#### 3.0.2.5 Fine-Tuning Llama 3.2

Fine-tuning was conducted using Google Colab with a Tesla T4 GPU, and 14.748 GB memory. A resource-efficient fine-tuning strategy was adopted using Low-Rank (LoRA) [65], which reduces the number of trainable parameters by introducing rank decomposition matrices in transformer layers. This significantly lowers computational overhead while maintaining model performance. Table 3.6 shows the fine-tuning parameters and their respective. To improve instruction adherence, each training instance included an End-of-Sentence (EOS) token, enhancing response coherence.

AdamW was selected as the optimizer due to its well-established performance in training transformer-based models, particularly in scenarios involving supervised fine-tuning [166]. The 8-bit precision setting was adopted to reduce memory usage without significant degradation in accuracy, enabling training on modest hardware (e.g., Tesla T4 GPU) [137].

The learning rate of 2e-5 was informed by prior work in low-resource fine-tuning of DSLLMs and was found to offer a stable trade-off between convergence speed and generalization [127, 158]. LoRA settings (rank = 8, alpha = 16, dropout = 0.1) were selected to enable efficient parameter updates with minimal overhead, consistent with best practices in low-rank adaptation studies [65, 146].

A batch size of 8 was used to ensure stable gradient updates within the memory limits of the training environment, while 60 training iterations were sufficient to observe convergence on the Alpaca-formatted instruction-following dataset.

#### 3.0.2.6 VulnFixAI Technical Architecture

VulnFixAI's technical architecture integrates fine-tuned Llama 3.2 (3B) model within a full-stack application designed for automated vulnerability detection and repair in Java code. The system runs locally using Ollama [112] on a MacBook Pro with Apple M3 Pro chip, 18GB RAM, and macOS 15.3.1. The frontend, developed with React [100], provides a dynamic user interface where developers upload Java snippets and interact with highlighted results. These results include vulnerable code snippets, their corresponding CWE-ID (e.g., CWE-23), a detailed explanation of the vulnerability's cause, and a suggested fix. If the developer agrees with the fix, they can apply it with a single click, streamlining the remediation process. The backend, implemented as a Java-based API, interfaces with the Llama model running on Ollama to process snippets, identify vulnerabilities, and generate repairs.

# CHAPTER FOUR

# EVALUATION

This chapter outlines the evaluation plan designed to assess the effectiveness of the proposed refactoring techniques and the VulnFixAI tool.

### 4.0.1 Refactoring Techniques

The effectiveness of the proposed refactoring techniques is evaluated by applying them to twenty one open-source projects selected from GitHub, covering a range of domains and sizes. These projects were chosen for their focus on security and reliability with active communities on development practices. The selection was facilitated by utilizing GitHub's search filters to find projects with a high number of stars and forks, which suggest frequent use and ongoing maintenance. Each project was preprocessed by cloning the repository to a local machine, setting up the necessary configurations, and installing required libraries and dependencies, such as the Java JDK. This preparation ensured that each project was properly configured for thorough analysis and refactoring. The projects and their details are summarized in Table 4.1.

Table 4.1
Open Source Projects Used in Evaluation

| **Domain** | **Project Name** | **Description** | **LOC** | **Year** | **Ref** |
|---|---|---|---|---|---|
| Distributed Systems | ZooKeeper | Apache ZooKeeper is used for coordination in distributed systems, managing configuration and synchronization, and providing group services. | 54090 | 2008 | [46] |
| | BookKeeper | Apache BookKeeper is a distributed storage system designed for high-throughput and low-latency data logging, often used in real-time applications. | 145983 | 2011 | [46] |

**Table 4.1 – Continued**

| Domain | Project Name | Description | LOC | Year | Ref |
|---|---|---|---|---|---|
| | Ignite | Apache Ignite is a distributed, in-memory database for high-performance computing. | 702672 | 2015 | [45] |
| Medical | OpenMRS | OpenMRS is a platform designed to create medical record systems aimed at improving healthcare delivery. | 86897 | 2012 | [115] |
| | HAPI FHIR | HAPI FHIR is a Java implementation of the HL7 FHIR standard for healthcare data interoperability. | 234791 | 2014 | [60] |
| | Weasis | Weasis is a free DICOM viewer for medical images, available as standalone and web-based software for healthcare professionals and patients. | 91644 | 2019 | [60] |
| E-commerce | Broadleaf Commerce | An e-commerce framework based on the Spring Framework, designed for building customizable and scalable online stores. | 141842 | 2013 | [19] |
| | Shopizer | Shopizer is an open-source Java eCommerce platform offering headless architecture and RESTful API. | 52844 | 2015 | [129] |
| | Apache OFBiz | Apache OFBiz is an open-source enterprise resource planning (ERP) system offering integrated solutions for CRM, e-commerce, supply chain, and manufacturing management. | 317988 | 2020 | [110] |
| Education | Apache OpenMeetings | Apache OpenMeetings provides video conferencing, instant messaging, a whiteboard, collaborative document editing, and other groupware tools. | 51862 | 2020 | [9] |
| | Teammates | TEAMMATES is a free, cloud-based tool for educators to manage student peer evaluations and feedback, used globally by numerous universities. | 183506 | 2017 | [141] |
| | OpenOlat | OpenOlat is a web-based learning management system (LMS) offering comprehensive features for online teaching, learning, assessment, and communication. | 9882 | 2011 | [116] |
| Financial | Apache Fineract | Apache Fineract is an open-source software platform for financial services. It offers a mature, reliable, and robust core banking solution with open APIs. | 622182 | 2018 | [42] |
| | Portfolio Performance | Portfolio Performance is an open-source tool for tracking and analyzing investment portfolio performance across multiple asset types, including stocks and cryptocurrencies. | 419677 | 2013 | [122] |

**Table 4.1 – Continued**

| Domain | Project Name | Description | LOC | Year | Ref |
|---|---|---|---|---|---|
| | bitcoinj | The bitcoinj library, is a Java implementation of the Bitcoin protocol, allows wallet management and transaction processing without a local Bitcoin Core installation. | 92413 | 2014 | [18] |
| IoT | OpenRemote | OpenRemote is an IoT platform offering comprehensive device management, automation, analytics, and customization features for building complete IoT solutions. | 141489 | 2016 | [118] |
| | ThingsBoard | ThingsBoard is an open-source IoT platform for device management, data collection, processing, and visualization. | 415857 | 2016 | [144] |
| | Eclipse Kura | Eclipse Kura is IoT gateways, offering web-based configuration, extensible APIs, and streamlined data management from devices to cloud. | 429491 | 2014 | [39] |
| AI | Smile | Smile is a comprehensive Java and Scala library for machine learning, NLP, and data analysis, offering high-performance algorithms and visualization tools. | 226120 | 2016 | [131] |
| | Stanford CoreNLP | Stanford CoreNLP is toolkit for multilingual natural language processing, offering integrated text analysis tools from basic tokenization to advanced syntactic and semantic parsing. | 826665 | 2020 | [28] |
| | Encog | Encog is a Java, C# machine learning framework for neural networks and genetic algorithms, developed in 2008 and widely used in academic research. | 168413 | 2014 | [41] |

To identify vulnerabilities in these projects, the Snyk plugin in IntelliJ IDE [133, 75] was utilized, which is renowned for its integration with Snyk's extensive vulnerability database [134]. The tool has demonstrated its effectiveness in identifying security weaknesses and misconfigurations in Maven [10] and npm projects [138]. It provides a detailed categorization of issues based on their type and severity, facilitating a comprehensive understanding of the security challenges faced by the analyzed projects in this work. The evaluation process involved the following steps.

1. Conduct a comprehensive scan of each project using the Snyk plugin. This initial scan serves as a baseline to understand the existing vulnerabilities within the projects, detailing their nature, severity, and location within the codebase.

2. Apply specific refactoring techniques to identified vulnerabilities. This step requires careful implementation of changes which are documented for analysis of the effectiveness of refactoring techniques.

3. Re-scan the projects using the Snyk plugin to assess the impact of the refactoring on the vulnerabilities. This is compared with the initial scan to determine how many vulnerabilities were addressed and if any new issues were introduced as a result of the changes.

4. Analyze the pre- and post-refactoring results to quantify the effectiveness of the refactoring techniques. This step analyzes trends such as reductions in the number of high-severity vulnerabilities, changes in the types of vulnerabilities, and overall improvements in code quality.

5. Use the insights gained and challenges encountered from the evaluation to refine refactoring techniques.

#### 4.0.2 VulnFixAI

To evaluate VulnFixAI's effectiveness, a benchmark was conducted using twenty real-world Java projects from the GitHub Advisory Database, each tied to confirmed Common Vulnerabilities and Exposures (CVE) entries and validated by GitHub's Security Advisory team. Selected for their security-critical relevance, active maintenance, and diversity (66 to 16,850 lines of code), these projects span varied scopes, as listed in Table 4.2. The evaluation focused on three key metrics (1) vulnerability identification rate, (2) fix rate, and (3) overall effectiveness rate, measured across three vulnerability cate-

gories ITV, LOWV, and LOIS. These categories align with the CWE IDs addressed by the refactoring techniques WVR, OSR, and TCVR, as detailed in Section 3.0.1.2.

Each project was preprocessed by cloning only the affected repository versions explicitly containing the documented CVEs, as listed in Table 4.2. This ensured that the analysis focused on the specific vulnerable code states associated with each CVE, configured with required dependencies (e.g., Java JDK, Maven, Gradle).

Table 4.2
Open Source Projects Used in Evaluation

| Project Name | CVE-ID | CWE-ID | Description | Affected Versions | REF |
|---|---|---|---|---|---|
| eclipse-vertx | CVE-2019-17640 | CWE-22, CWE-23 | Vert.x core offers low-level, event-driven, non-blocking functionalities like HTTP, TCP, and file system access for building reactive applications. | >= 3.0.0, < 3.9.4 | [47] |
| Apache Flink | CVE-2020-17518 | CWE-22, CWE-23 | Apache Flink is an open-source framework for real-time stream and batch data processing. | >= 1.5.1, < 1.11.3$t$ | [44] |
| OpenTSDB | CVE-2020-35476 | CWE-78 | OpenTSDB is a distributed time series database built on HBase, designed for storing and serving large-scale metrics. | ¡= 2.4.0 | [140] |
| Apache Hadoop | CVE-2022-25168 | CWE-78, CWE-88 | Apache Hadoop enables distributed processing of large data sets across clusters using the MapReduce programming model and HDFS for storage. | >= 2.0.0, < 2.10.2, >= 3.0.0 $-$ $alpha$, < 3.2.4, >= 3.3.0, < 3.3.3 | [57] |
| Netty | CVE-2022-41915 | CWE-113, CWE-436 | Netty is a Java-based asynchronous event-driven network application framework for rapid development of high-performance protocol servers and clients. | >= 4.1.83.$Final$, < 4.1.86.$Final$ | [103] |
| Undertow | CVE-2018-1067 | CWE-113 | Undertow is a flexible Java web server offering blocking and non-blocking APIs, enabling composable architectures and full servlet 4.0 support. | <= 7.1.1.$GA$ | [27] |
| wire | CVE-2021-41193 | CWE-134 | Wire is a secure messaging platform; this CVE relates to its use of pre-built Google WebRTC binaries in AVS. | <= 7.1.1.$GA$ | [155] |
| Apache Dubbo | CVE-2021-36161 | CWE-134 | Apache Dubbo is a high-performance Java-based RPC framework offering microservices communication and service governance. | < 7.1.12 | [37] |
| Apache NiFi | CVE-2018-17195 | CWE-319, CWE-863 | Apache NiFi automates data flow between systems, providing a user-friendly interface for data ingestion, routing, and transformation. | < 2.7.13 | [106] |
| James | CVE-2022-45935 | CWE-200, CWE-319, CWE-668 | Apache James is a modular Java-based mail server platform, allowing custom email solutions with extensible components. | >= 1.0.0, <= 1.7.1 | [74] |
| Infinispan | CVE-2019-10174 | CWE-470 | Infinispan is an open-source in-memory key/value data store and cache, offering high availability and scalability. | <= 8.2.11.$Final$, >= 9.0.0.$Final$, <= 9.4.16.$Final$ | [61] |
| HyperSQL | CVE-2022-41853 | CWE-470 | HyperSQL Database (HSQLDB) is a relational database management system written in Java, offering in-memory and disk-based tables. | < 2.7.1 | [69] |
| Apache Hive | CVE-2022-41137 | CWE-502 | Apache Hive provides SQL-like querying for large datasets stored in Hadoop, facilitating data warehousing and analysis. | 4.0.0 $-$ $alpha$ $-$ 1 | [63] |
| Openmeetings | CVE-2024-54676 | CWE-502 | OpenMeetings is an Apache-licensed platform for video conferencing, instant messaging, and collaborative document editing. | >= 2.1.0, < 8.0.0 | [114] |
| Eclipse GlassFish | CVE-2024-9329 | CWE-233, CWE-601 | Eclipse GlassFish is a Jakarta EE-compliant application server, offering implementations of all Jakarta EE APIs. | >= 2.1.0, < 8.0.0 | [38] |
| Keycloak | CVE-2024-8883 | CWE-601 | Keycloak is an open-source identity and access management solution, supporting single sign-on with OAuth2, OpenID Connect, and SAML. | < 7.0.17 | [80] |
| Hugegraph-toolchain | CVE-2024-27347 | CWE-918 | HugeGraph-Toolchain integrates utilities for HugeGraph, including over five main modules for enhanced graph data management. | <= 22.0.12, >= 23.0.0, <= 24.0.7, >= 25.0.0, <= 25.0.5 | [68] |
| Apache CXF | CVE-2024-28752 | CWE-918 | Apache CXF is an open-source services framework facilitating the development of SOAP and RESTful web services. | >= 1.0.0, < 1.3.0 | [32] |
| FitNesse | CVE-2024-39610 | CWE-79 | FitNesse is a web server-based tool for specifying and verifying application acceptance criteria through collaborative testing. | < 3.5.8, >= 3.6.0, < 3.6.3, >= 4.0.0, < 4.0.4 | [43] |
| OpenRefine | CVE-2024-47882 | CWE-79, CWE-81 | OpenRefine is a Java-based tool for cleaning and transforming messy data, enhancing data quality and consistency. | < 3.8 | [117] |

# CHAPTER FIVE

# RESULTS

This chapter presents the results of evaluating the proposed refactoring techniques and the VulnFixAI tool.

### 5.0.1 Analysis of Existing Vulnerabilities

To identify vulnerabilities in these projects, the Snyk plugin in IntelliJ IDE [133, 75] was employed, which is renowned for its integration with Snyk's extensive vulnerability database [134]. The tool has demonstrated its effectiveness in identifying security weaknesses and misconfigurations in Maven [10] and npm projects [138]. It provides a detailed categorization of issues based on their type and severity, facilitating a comprehensive understanding of the security challenges faced by the analyzed projects in this work. The evaluation process involved the following steps.

1. Conduct a comprehensive scan of each project using the Snyk plugin. This initial scan serves as a baseline to understand the existing vulnerabilities within the projects, detailing their nature, severity, and location within the codebase.

2. Apply specific refactoring techniques to identified vulnerabilities. This step requires careful implementation of changes which are documented for analysis of the effectiveness of refactoring techniques.

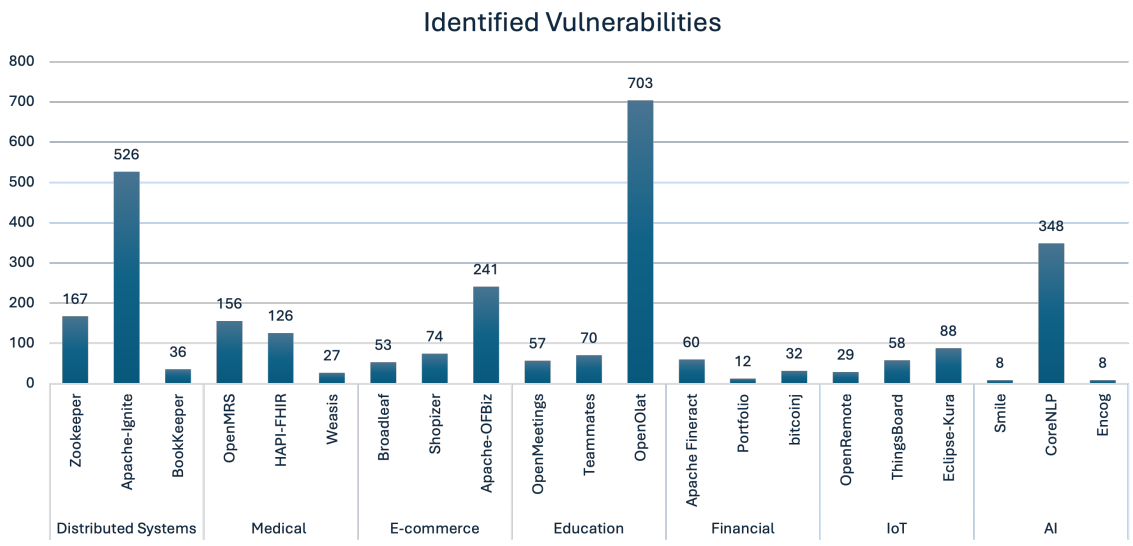
3. Re-scan the projects using the Snyk plugin to assess the impact of the refactoring on the vulnerabilities. This is compared with the initial scan to determine how many vulnerabilities were addressed and if any new issues were introduced as a result of the changes.

4. Analyze the pre- and post-refactoring results to quantify the effectiveness of the refactoring techniques. This step analyzes trends such as reductions in the number of high-severity vulnerabilities, changes in the types of vulnerabilities, and overall improvements in code quality.

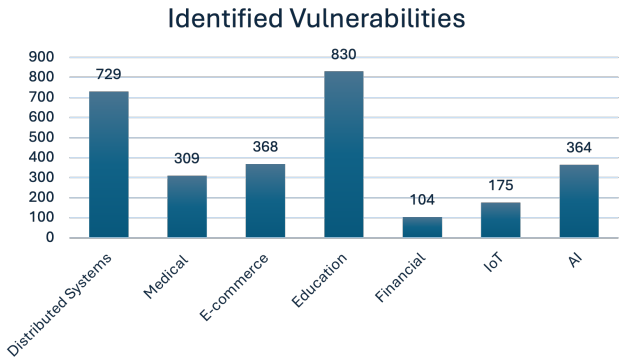
5. Use the insights gained and challenges encountered from the evaluation to refine refactoring techniques.

Figure 5.1 presents the number of identified vulnerabilities by project and domain. Graph (a) shows significant variations among projects. Notably, OpenOlat in the Education domain exhibits an exceptionally high count of 703 vulnerabilities, indicating a critical area for security enhancements. On the other hand, Smile and Encog in the AI domain present the lowest vulnerability counts, each with only 8 identified issues. Graph (b) quantifies vulnerabilities by domain, showing that the Education domain has the highest number, totaling 830, while the Financial domain has the fewest with only 104. This data suggests a potential correlation between the application domain and associated security risks, which suggests tailored security strategies for each domain. The average number of identified vulnerabilities is 137 for projects and 411 for domains, respectively.

Figure 5.2 presents the distribution of vulnerability categories within each domain. It reveals that across domains, LOIS, LOWV, and ITV are dominant, reflecting the importance of input sanitization, input validation, and data integrity. Notably, the Financial domain shows predominance by ITV, accounting for 74%, while the AI and Distributed Systems domains demonstrate a dominance of LOWV (45% and 52%, respectively), along with involvement in a more diverse range of vulnerabilities. On the other hand, the Education domain exhibits significant susceptibility to input-related vulnerabilities, as indicated by LOIS accounting for 45%. Other categories such as LOBC, INH, ITH, and IEF, occur less frequently across all domains with percentages ranging from 0% to 4%,




(a) Identified Vulnerabilities by Project




(b) Identified Vulnerabilities by Domain

Figure 5.1: Identified Vulnerabilities

indicating that these issues are either less common or more consistently addressed during development.

#### 5.0.1.1 Applying Refactoring Techniques

After analyzing the existing vulnerabilities in the projects, the proposed refactoring techniques were manually applied to address them..

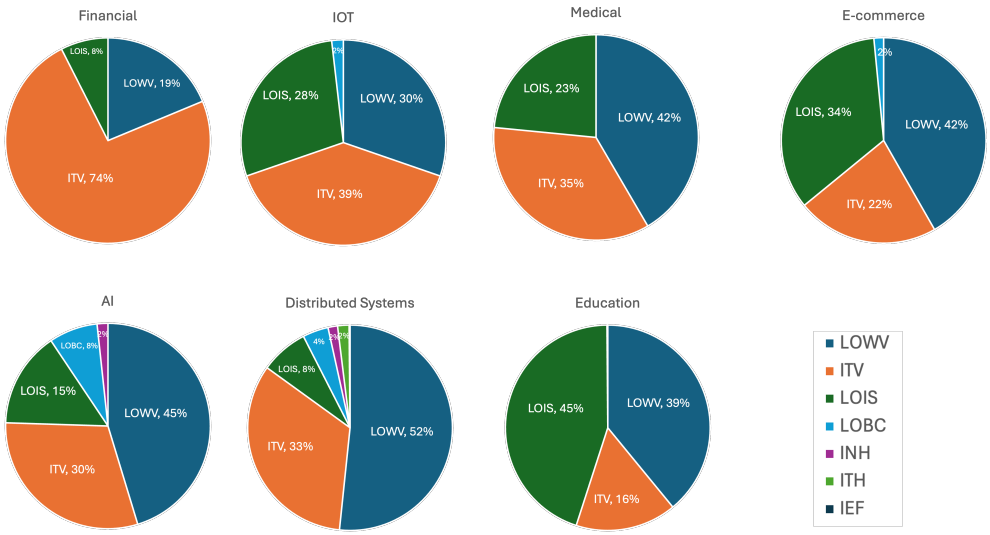



Figure 5.2: Distribution of Vulnerability Categories by Domains

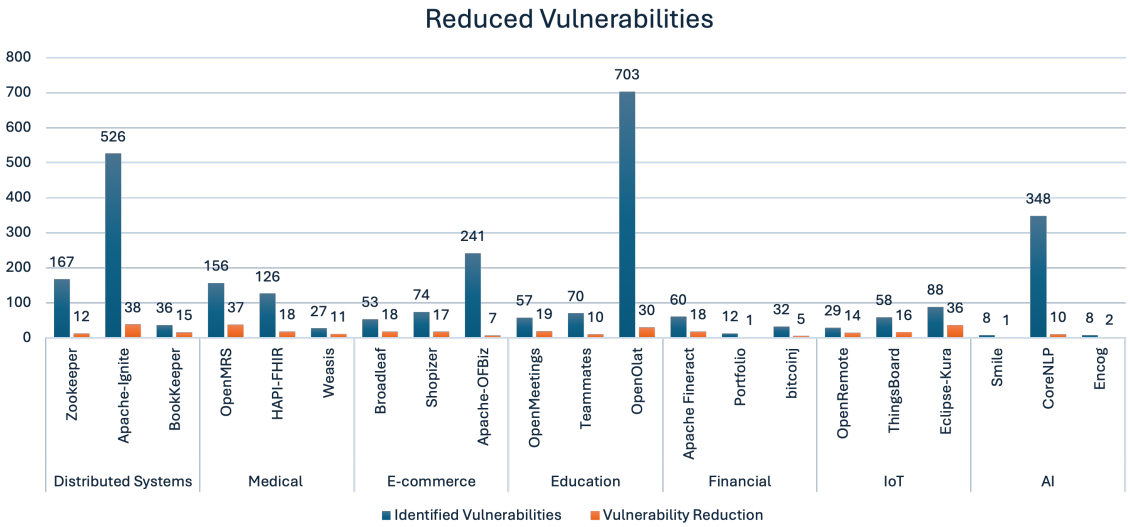
For low-complexity vulnerabilities, such as LOBC and IAC, refactoring was relatively straightforward. Techniques like BSR and SACR were applied by introducing boundary checks and using safer array-copying methods. These modifications were easily integrated with minimal changes to project logic. However, higher-complexity vulnerabilities, such as LOIS (e.g., XSS and SQL injection) and ITV (e.g., improper trust verification), required significant manual effort. Refactoring techniques such as OSR and TCVR needed deeper context analysis to ensure input sanitization and secure certificate validation were correctly applied. Similarly, LOWV, which deals with insufficient input validation, required restructuring validation logic to enforce strict allowlist-based filtering.

Figure 5.3 displays the outcomes of this application. Graph (a) illustrates the number of vulnerabilities remaining post-application, while graph (b) depicts the reduction percentages. According to the results, Apache-OFBiz and CoreNLP exhibited the greatest reduction, both at 97%, decreasing from 241 to 7 and from 348 to 10, respectively. The significant reduction in Apache-OFBiz is due to the comprehensive removal of the

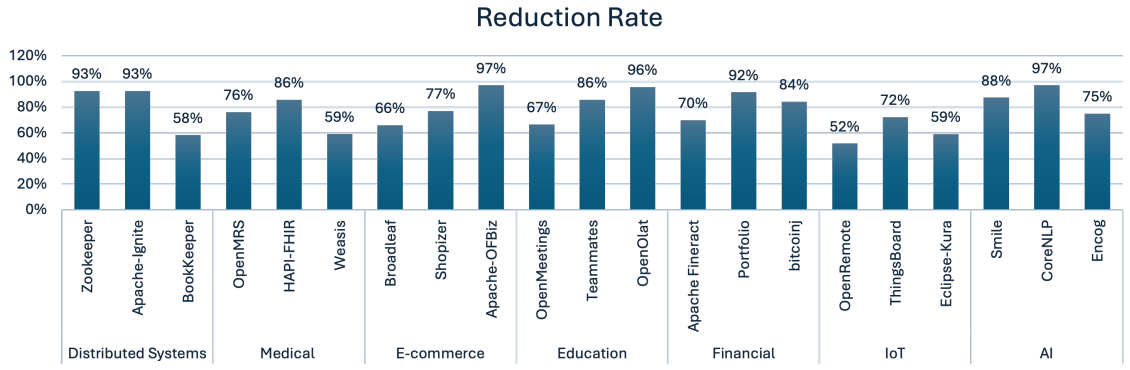
dominant vulnerabilities within the LOWV, ITV, and LOBC categories, which accounts for 95% (229/241) of identified vulnerabilities. Similarly, CoreNLP's substantial reduction is attributed to eliminating vulnerabilities across the LOWV, ITV, LOIS, and LOBC categories, achieving a 97% reduction (338/348). On the other hand, OpenRemote exhibited the smallest reduction at 52%, decreasing from 29 to 14. This modest decrease is due to vulnerabilities related to JUnit tests and dependencies with third-party libraries, which were not addressed in this work as they are not part of the production code. Similar observations are made for other projects (e.g., BookKeeper, Weasis, Eclipse-Kura) that present small reductions. Overall, the average number of remaining vulnerabilities across all projects has been reduced to 335 from 2,879, representing an 88% reduction.

Figure 5.4 shows the application results by domain. Graph (a) shows the remaining vulnerabilities after applying refactoring techniques and graph (b) presents the reduction percentage. Per the data, the AI domain demonstrates the most reduction at 96% down to 13 from 364, while the IoT domain shows the least reduction at 62% down to 66 from 175.

The high reduction of the AI domain is due to addressing the large number of vulnerabilities with LOWV, ITV, LOIS, and LOBC which account for 94% (345/364). The modest reduction in the IoT domain is attributed to the numerous vulnerabilities present in JUnit tests and third-party library dependencies (e.g., io.moquette, log4j), accounting for 32% (52/175), which were not addressed in this study as they are not part of the production code. A similar observation is made for the Financial domain which exhibits the next smallest reduction. After applying refactoring techniques, the average number of remaining vulnerabilities for domains is reduced to 48 from 137, representing an 88% reduction. Table A.1, Table B.1, and Table C.1 in the Appendix detail the remaining vulnerabilities by project, domain, and category. These tables reveal that approximately 58% (194/335) of the remaining vulnerabilities are related to JUnit tests, which intentionally include hard-



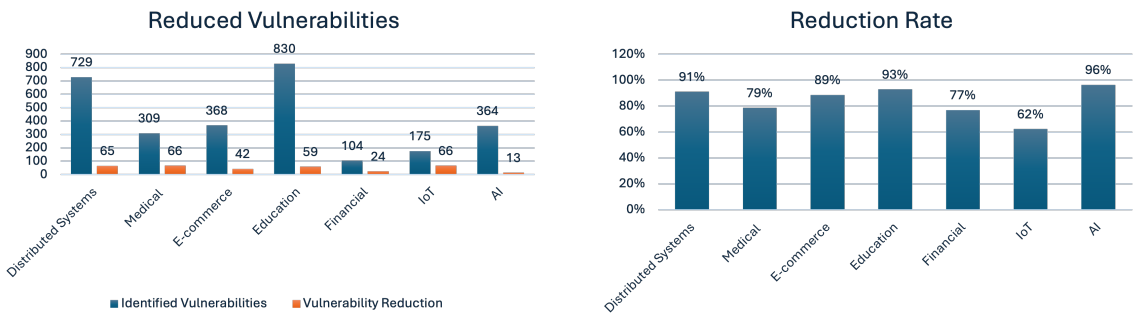

(a) Reduced Vulnerabilities by Project




(b) Reduction Rate

Figure 5.3: Reduced Vulnerabilities by Project

coded credentials or other vulnerabilities for testing purposes but are not present in the production code. These tests often depend on mocking frameworks and other testing libraries that may introduce vulnerabilities (e.g., CWE-502). The remaining vulnerabilities (141/335), accounting for 42% of the total, originate from dependencies with third-party libraries (e.g., io.moquette, com.sun.jersey, mysql).




(a) Reduced Vulnerabilities by Domain

(b) Reduction Rate

Figure 5.4: Reduced Vulnerabilities by Domain

These results demonstrate the effectiveness and broad applicability of the proposed refactoring techniques in addressing vulnerabilities in diverse contexts. The high reduction percentages, particularly in complex domains like AI and Distributed Systems, suggest that these techniques have potential to improve software security practices across the industry.

#### 5.0.1.2 Dependency Analysis

Vulnerabilities are multifaceted, leading to different solutions depending on the perspectives from which they are interpreted. This implies that refactoring techniques might have dependencies that should be considered together when they are applied. This section discusses the dependencies among the refactoring techniques introduced in Section 3.0.1.2. Figure 5.5 illustrates their dependencies.

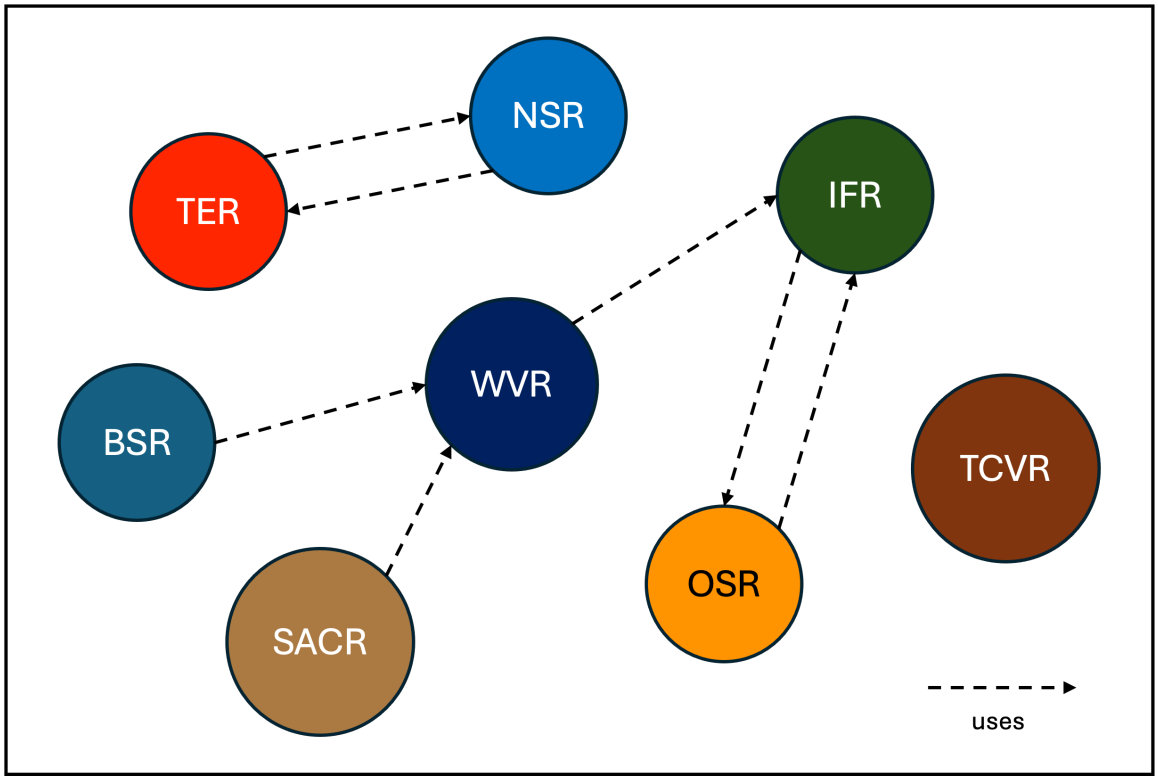



Figure 5.5: Dependencies among Refactoring Techniques

- SACR focuses on preventing buffer overflow vulnerabilities by ensuring that the size of the data being copied does not exceed the array's boundaries. WVR can complement this by verifying that all incoming data meets specified criteria before it is processed, ensuring that only valid, expected data sizes are allowed. This involves checking the length of data against a predefined list of acceptable lengths, thereby preventing any attempt to copy oversized or undersized arrays that could lead to security risks. For example, for the vulnerability in Listing 3.3, WVR and SACR can be applied together as demonstrated in Listing 5.1. In the code, WVR introduces an $if$ statement to check whether the length of $userData$ exceeds the capacity of the $dataArray$. If it does, an exception is thrown, ensuring that only data that fits into the array is processed, thereby preventing potential overflows before attempting to copy the data. Subsequently, SACR employs $System.arraycopy$ for safer copying, which explicitly requires the length of data to be copied and ensures that data does not write beyond the end of the destination array. This method offers an alternative solution to the solution in Listing 3.4.

```
1  public void storeUserData(String userData) {
2    char[] dataArray = new char[50];
3    if (userData.length() > dataArray.length) { // <—— Fix by WVR
4      throw new IllegalArgumentException("Input data too large.");
5    }
6    System.arraycopy(userData.toCharArray(), 0, dataArray, 0, userData.length()); // <—— Fix by SACR
7  }
```

Listing 5.1: After WVR and SACR

- BSR secures buffer management by introducing checks that prevent buffer overflows. BSR may utilize WVR to ensure that data processed through buffers adheres to specific, predefined safety criteria. For instance, WVR can validate the lengths of data or the types of inputs being handled, ensuring that they are within safe operational parameters before BSR applies boundary checks. This integration prevents BSR from processing any data that could potentially violate buffer boundaries. By

using WVR, BSR ensures that only validated, safe data is allowed to pass through critical sections of code, reducing the risk of security breaches related to buffer management. For example, the vulnerability in Listing 3.1 can be addressed by WVR and BSR together as shown in Listing 5.2. In the code, WVR is applied before BSR to check if the source array's length exceeds the destination array's capacity. This validation ensures that no attempt is made to copy more elements than the destination can hold, preventing any possibility of buffer overflow right from the start.

```
1  public void dataCopy(char[] source) {
2    char[] destination = new char[10];
3    if (source.length > destination.length) { <—— Fix by WVR
4      throw new IllegalArgumentException("Source array too large");
5    }
6    for (int i = 0; i < source.length; i++) {
7      if (i >= destination.length) { // <—— Fix by BSR
8        throw new ArrayIndexOutOfBoundsException("Buffer Overflow Prevented");
9      }
10     destination[i] = source[i];
11   }
12 }
```

Listing 5.2: After BSR and WVR

- OSR and IFR can be used together to support each other. OSR, which focuses on sanitizing output to prevent security vulnerabilities such as XSS, can utilize IFR to ensure that the error messages provided to users are not only safe but also informative and clear. Conversely, IFR, which aims to refine feedback provided to users based on their inputs, can leverage OSR to sanitize any user input displayed back on the user interface, ensuring that no harmful scripts are executed. This integration helps in maintaining a secure feedback loop where user inputs are checked, sanitized, and then displayed safely, enhancing both the security and the user-friendliness of the application. For example, the vulnerability in Listing 3.5 can be addressed by OSR and IFR together as demonstrated in Listing 5.3. In the code, IFR checks if the username is empty and provides a clear, direct error mes-

sage if necessary. This message does not expose any system details but informs the user of the specific issue with their input.

```
1  public void generateWelcomeMessage(HttpServletRequest request, HttpServletResponse response) throws IOException {
2    String username = request.getParameter("username");
3    if (username == null || username.isEmpty()) { // <—— Fix by IFR
4      response.getWriter().println("Error: Username cannot be empty.");
5    } else {
6      String sanitizedUsername = Encode.forHtml(username);
7      String htmlOutput = "<p>Welcome, " + sanitizedUsername + "</p>"; // <—— Fix by OSR
8      response.getWriter().println(htmlOutput);
9    }
10 }
```

Listing 5.3: After OSR and IFR

- TER and NSR can be strategically aligned to complement each other. TER ensures that data types are correctly managed and enforced throughout the application, preventing type mismatches that could lead to vulnerabilities. NSR, focusing on the safety of numeric operations, ensures that all arithmetic operations do not exceed the limits of the data types involved, thus preventing errors like overflows or underflows. When integrated, NSR can benefit from the strong type guarantees provided by TER, ensuring that arithmetic operations are performed on data types that are well-defined against type-based errors. Conversely, TER can utilize the safeguards established by NSR to ensure that any enforced data types are not only correctly implemented but also secure from numeric vulnerabilities, enhancing both data integrity and operational safety. For example, the vulnerability in Listing 3.11 can be addressed by TER and NSR together as shown in Listing 5.4. In the code, TER is applied through the conversion of *String* to *int* using $Integer.parseInt()$. This ensures that the input strings are actually valid integers. Then, NSR is applied to prevent potential overflows in the operation.

```
1   public int add(String strA, String strB) {
2     try {
3       int a = Integer.parseInt(strA);  // <—— Fix by TER
4       int b = Integer.parseInt(strB);
5       if (Integer.MAX_VALUE — a < b) { // <—— Fix by NSR
6         throw new ArithmeticException("Integer overflow");
7       }
8       return a + b;
9     } catch (NumberFormatException e) {
10      throw new IllegalArgumentException("Input must be a valid integer", e);
11    }
12  }
```

Listing 5.4: After TER and NSR

- WVR focuses on validating user inputs against a set of predefined acceptable patterns or values, ensuring that only safe and expected data is processed by the application. IFR, which aims to provide clear and useful feedback to users, can use the results of WVR's validation to inform users about why certain inputs were rejected and how to correct them. For example, consider the example in Listing 5.5. In the code, there is no validation for the email parameter, making the code susceptible to various forms of input-based attacks, such as SQL injection or other forms of malicious input exploitation, which introduces an LOWV vulnerability. Also, the code does not provide any handling of exceptions or any feedback to the user about the expected format of the email or what might be wrong with their input if it fails any subsequent processing checks, which introduces an IEF vulnerability. These vulnerabilities can be addressed by WVR and IFR together as shown in Listing 5.6. In the refactored code, WVR introduces a regular expression to validate the *email* format, ensuring that only properly formatted emails are accepted. This validation helps prevent issues such as SQL injection (if the email is used in database queries) or other forms of injection and format errors. If the email does not match the required format, IFR provides feedback to the user, explaining exactly what is wrong with

the input. This clarity helps users correct their inputs effectively without guessing what might be wrong.

```
1  public void processForm(HttpServletRequest request, HttpServletResponse response) throws IOException {
2      String email = request.getParameter("email");
3  }
```

Listing 5.5: Before WVR and IFR

```
1  public void processForm(HttpServletRequest request, HttpServletResponse response) throws IOException {
2      String email = request.getParameter("email");
3      if (email == null || !email.matches("^[a-zA-Z0-9_+&*-]+(?:\\.[a-zA-Z0-9_+&*-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,7}$")) { //
                <-- Fix by WVR
4          response.getWriter().println("Invalid email format. Please provide a valid email address."); // <-- Fix by IFR
5          return;
6      }
7      response.getWriter().println("Email processed successfully.");
8  }
```

Listing 5.6: After WVR and IFR

### 5.0.2 VulnFixAI Results

VulnFixAI's performance was benchmarked against four state-of-the-art language models, ChatGPT-4, Claude 3.5 Sonnet, Gemini 2.0 Flash and the non fine-tuned Llama 3.2 (3B). These baselines were chosen due to their prominence in code-related tasks and their ability to process natural language instructions paired with code snippets, similar to VulnFixAI's approach. Unlike traditional static analysis tools (e.g., SonarQube, Fortify) primarily rely on predefined rules and pattern matching, which limits their flexibility in handling novel or context-dependent vulnerabilities. For fairness, each model received identical inputs in the Alpaca format, consisting of instruction-input-output triplets (e.g., "Analyze the following code for CWE-79 and identify the vulnerable line" or "Apply a fix for CWE-601").

The evaluation metrics were defined as follows:

1. Identification Rate Per Category: The percentage of vulnerabilities correctly detected comparable to the total number of known vulnerabilities per category.

$$\text{Identification Rate} = \left( \frac{\text{Identified Vulnerabilities}}{\text{Total Vulnerabilities}} \right) \times 100$$

2. Fix Rate Per Category: The percentage of fix vulnerabilities.

$$\text{Fix Rate} = \left( \frac{\text{Fixed Vulnerabilities}}{\text{Identified Vulnerabilities}} \right) \times 100$$

3. Effectiveness Rate Per Category: The product of identification and fixed rates.

$$\text{Overall Effectiveness Rate Per Category} = \frac{\text{Fix Rate} \times \text{Identification Rate}}{100}$$

4. Overall Effectiveness Rate Across All Categories

$$\text{Overall Effectiveness Rate} = \frac{\sum(\text{Effectiveness Rate}_i \times \text{Total Vulnerabilities}_i)}{\sum \text{Total Vulnerabilities}_i}$$

Where:

$$\text{Effectiveness Rate}_i = \frac{\text{Identification Rate}_i \times \text{Fix Rate}_i}{100}$$

#### 5.0.2.1 Analysis Vulnerability By Categories

As shown in Figure 5.6, VulnFixAI achieved a remarkable identification rate of 100% across all three vulnerability categories LOWV, ITV, and LOIS. This result highlights VulnFixAI's superior accuracy and robustness in detecting vulnerabilities compared to state-of-the-art baseline models. Among the baseline models, ChatGPT-4 demonstrated the highest performance, achieving 85% for LOWV, 90% for ITV, and 85% for LOIS, followed closely by Claude 3.5 Sonnet with identification rates of 83%, 86%, and 83%, respectively. Gemini 2.0 Flash showed slightly lower detection capability, achiev-

ing 82%, 81%, and 82% across the three categories, while the non-fine-tuned Llama 3.2 (3B)exhibited the lowest performance with 72%, 76%, and 85%. These results emphasize the impact of fine-tuning domain-specific models for vulnerability detection. Unlike general-purpose LLMs, which rely on broad training datasets, VulnFixAI's targeted training on vulnerability-specific datasets significantly enhances detection accuracy. Furthermore, the substantial performance gap between VulnFixAI and the baseline models demonstrates the effectiveness of integrating refactoring techniques (e.g. WVR, OSR, and TCVR) with AI-driven pattern recognition.

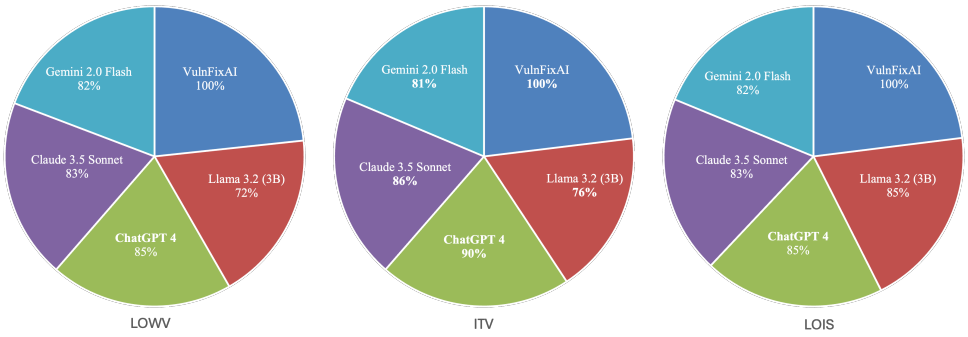



Figure 5.6: Identification Rate By Category

As shown in Figure 5.7 , VulnFixAI achieved high fix rates across all three vulnerability categories with 89% for ITV and LOWV, and 88% for LOIS. These results demonstrate VulnFixAI's effectiveness in not only identifying but also applying structured refactoring-based fixes to mitigate software vulnerabilities. Compared to the baseline models, ChatGPT-4 exhibited strong performance, achieving 85% for LOWV, 86% for ITV, and 84% for LOIS, followed closely by Claude 3.5 Sonnet with fix rates of 83%, 84%, and 83%, respectively. Gemini 2.0 Flash maintained a consistent performance

of 82% across all categories, while the non-fine-tuned Llama 3.2 (3B) showed the lowest fix rates with 75% for LOWV, 77% for ITV, and 69% for LOIS. The performance gap between VulnFixAI and the baseline models highlights the advantage of integrating domain-specific refactoring techniques with AI-driven automation. Unlike general-purpose LLMs, which rely on broad training data and may not generalize well to vulnerability-specific fixes.

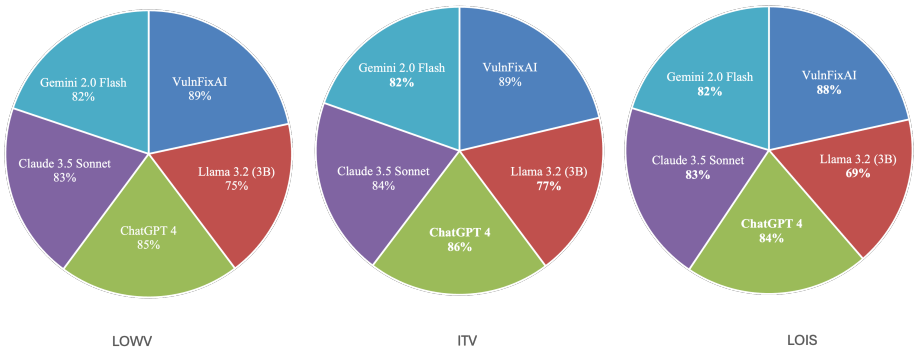



Figure 5.7: Fixes Rate By Category

As shown in Figure 5.8 , the overall effectiveness rate By Category, which combines both vulnerability identification and fix capabilities, highlights VulnFixAI's outperform over the baseline models across all three vulnerability categories. VulnFixAI achieved 89% effectiveness for LOWV and ITV, and 88% for LOIS, demonstrating its ability to accurately detect and systematically remediate vulnerabilities. Among the baseline models, ChatGPT-4 showed the highest effectiveness with 71% for LOWV, 77% for ITV, and 71% for LOIS, followed by Claude 3.5 Sonnet, which achieved 69%, 72%, and 59%, respectively. Gemini 2.0 Flashexhibited slightly lower effectiveness with 68%, 66%, and 59%, while the non-fine-tuned Llama 3.2 (3B) had the weakest performance, consis-

tently achieving 59% across all categories. These results reinforce the importance of fine-tuning AI models on domain-specific datasets to enhance both detection accuracy and automated remediation strategies. Unlike general-purpose LLMs, which often struggle with context-aware vulnerability fixes, VulnFixAI leverages AI-driven pattern recognition and automated refactoring techniques to achieve a more structured and effective approach.

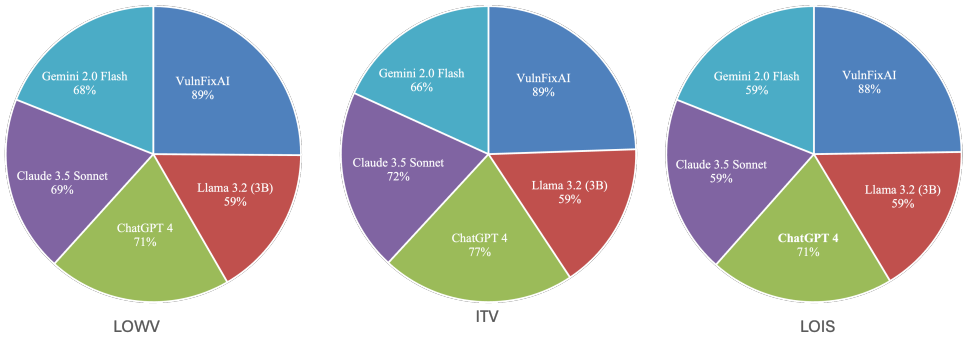



Figure 5.8: Overall Effectiveness Rate By Category

As shown in Figure 5.9, VulnFixAI achieved the highest overall effectiveness rate of 89%, significantly outperforming all baseline models. In comparison, ChatGPT-4 achieved 74%, followed by Claude 3.5 Sonnet at 70%, Gemini 2.0 Flash at 67%, and the non-fine-tuned Llama 3.2 (3B) at 59%. These results highlight VulnFixAI's superiority in both vulnerability detection and remediation, strengthening its role as a state-of-the-art solution for automated software security. This 15–30% improvement over baseline models validates VulnFixAI's integrated design. Unlike general-purpose LLMs that rely on broad, non-specialized training data, VulnFixAI leverages security-critical datasets, enabling precise vulnerability identification and systematic, structured fixes. This domain-

specific optimization significantly improves both detection accuracy and remediation effectivenes.

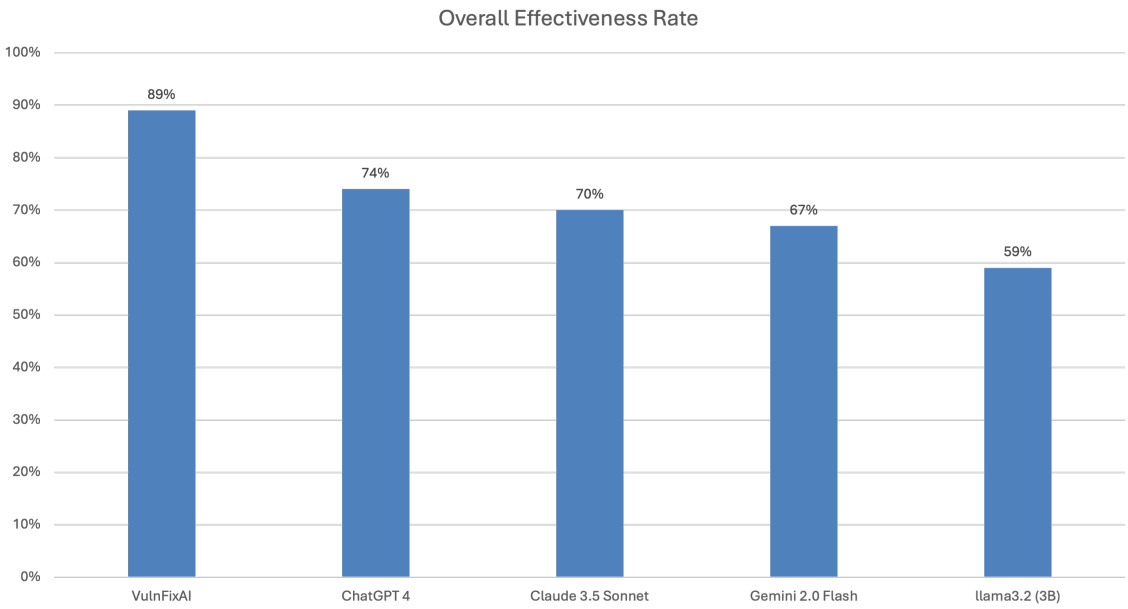
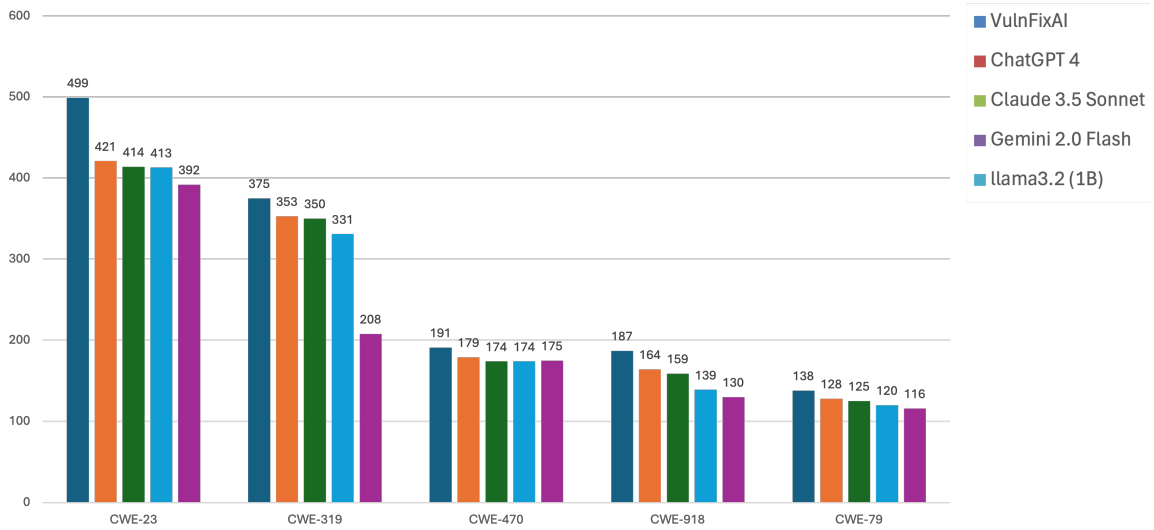



Figure 5.9: Overall Effectiveness Rate

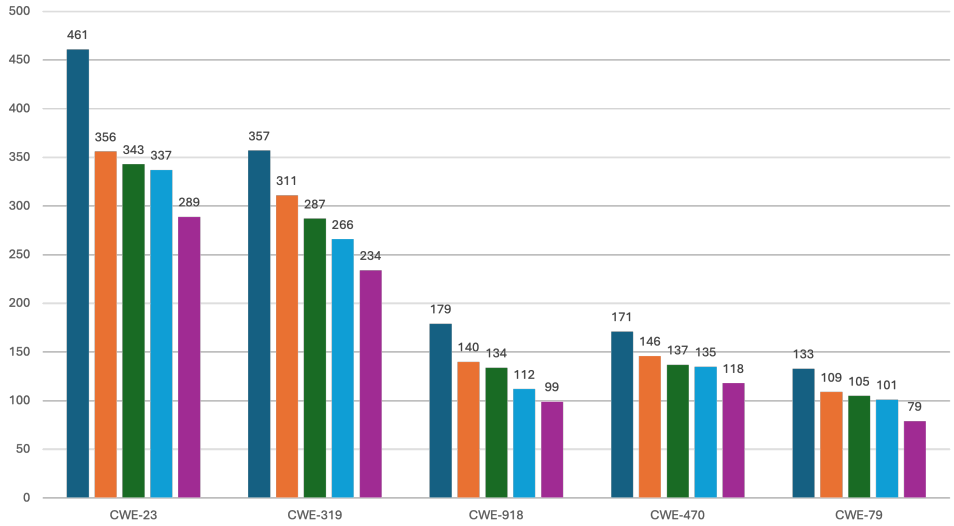
#### 5.0.2.2 Analysis Vulnerability By Projects

Figures( 5.10) illustrate the distribution of identified and fixed CWE vulnerabilities across multiple open-source projects. To provide a focused discussion, this section explains the five most frequently occurring vulnerabilities. The completed distribution of identification and fix results for all CWE types is provided in Appendix D(Tables D.1 and D.2). Furthermore, detailed project-wise results for top 5 CWE types are available in Appendix E (Tables E.1 and E.2).

As shown in Graph (a), VulnFixAI demonstrated remarkable performance, achieving a 100% vulnerability detection rate and identifying a total of 1897 vulnerabilities




(a) Identification distribution for the top five CWE types




(b) Fix distribution for the top five CWE types

Figure 5.10: Distribution of Identified and Fix Vulnerabilities Across Projects

across the evaluated projects. The most prevalent vulnerability was CWE-23 (Path Traversal), accounting for 24.3% (461 instances) with significant concentrations in Eclipse GlassFish (187 instances), Apache Hadoop (75), and Apache NiFi (60). CWE-319 (Cleartext Transmission of Sensitive Information) was the second most frequent, comprising 18.8% (357 instances), predominantly in Eclipse GlassFish (298), Apache NiFi (35),

and Undertow (9). Other notable vulnerabilities included CWE-470 (Use of Externally-Controlled Input to Select Classes or Code) at 9.0% (171 instances), primarily affecting Apache Hadoop (168), Eclipse GlassFish (12), and Keycloak (3). CWE-918 (Server-Side Request Forgery, SSRF) at 9.4% (179 instances) with the highest occurrences in Eclipse GlassFish (121), Apache CXF (23), and Apache Hadoop (16), and CWE-79 (Cross-Site Scripting, XSS) at 7.0% (133 instances), notably in Eclipse GlassFish (108), OpenMeetings (12), and Keycloak (8).

In comparison, ChatGPT-4 identified 1,914 vulnerabilities with CWE-23 as the most common at 22.0% (421), primarily in Eclipse GlassFish (181), Apache Hadoop (57), and Apache NiFi (46). CWE-319 followed at 18.5% (354 ), mainly in Eclipse GlassFish (292), Apache NiFi (27), and Undertow (7). CWE-470, CWE-918, and CWE-79 accounted for 9.3% (178), 8.4% (161), and 6.7% (128) of detections, respectively with similar project distributions to VulnFixAI but with slightly lower overall counts. Claude 3.5 Sonnet detected 1878 vulnerabilities with CWE-23 at 22.0% (414 ), primarily in Eclipse GlassFish (178), Apache Hadoop (56), and Apache NiFi (45). CWE-319 was second at 18.7% (351), focused in Eclipse GlassFish (290), Apache NiFi (31), and Undertow (7). CWE-470, CWE-918, and CWE-79 constituted 9.2% (173), 8.4% (157), and 6.7% (125) of detections, respectively with distributions mirroring those of other models but with reduced totals. Gemini 2.0 Flash identified 1827 vulnerabilities with CWE-23 as the most frequent at 22.6% (413), mainly in Eclipse GlassFish (180), Apache Hadoop (56), and Apache NiFi (45). CWE-319 followed at 18.2% (332 instances), predominantly in Eclipse GlassFish (276), Apache NiFi (26), and Undertow (7). CWE-470, CWE-918, and CWE-79 accounted for 9.5% (173), 7.5% (137), and 6.6% (120) of detections, respectively with Eclipse GlassFish and Apache Hadoop also showing the highest concentrations. LLlama 3.2 (3B) detected the fewest vulnerabilities, totaling 1797 with CWE-23 at 21.8% (392), primarily in Eclipse GlassFish (166), Apache Hadoop (54), and Apache

NiFi (44). CWE-319 was second at 11.6% (209), concentrated in Eclipse GlassFish (157), Apache NiFi (24), and Undertow (6). CWE-470, CWE-918, and CWE-79 accounted for 9.7% (174), 7.1% (128), and 6.4% (115) of detections, respectively with similar project distributions but lower overall detection rates.

Across all models, CWE-23 and CWE-319 consistently appeared as the most frequently detected vulnerabilities with detection rates ranging from 21.1% to 22.6% and 11.6% to 18.7%, respectively. This emphasizes a pervasive presence of -23 and CWE-319 vulnerabilities in Java-based open-source projects. Eclipse GlassFish and Apache Hadoop consistently exhibited the highest concentrations of these vulnerabilities with Eclipse GlassFish dominating CWE-319 detections (e.g., 298 instances by VulnFixAI) and Apache Hadoop leading in CWE-470 (e.g., 168 instances by VulnFixAI). VulnFixAI's superior performance identifying 1,897 vulnerabilities compared to 1797–1914 by the LLMs indicates its enhanced in vulnerability identification.

These findings highlight the crucial role of leveraging VulnFixAI's hybrid approach to vulnerability detection strategies to enhance software security across diverse open-source projects, offering valuable insights for prioritizing security efforts in high-risk codebases.

As shown in Graph (b), VulnFixAI demonstrated outstanding efficacy, resolving a total of 1711 out of 1,897 identified vulnerabilities across all projects. The most frequently addressed vulnerability was CWE-23, accounting for 26.94% (461) of total fixes, mostly in Eclipse GlassFish (187 fixes), Apache NiFi (56 fixes), and Apache CXF (50 fixes). Following closely, CWE-319 constituted 20.86% (357), primarily concentrated in Eclipse GlassFish (298), Apache NiFi (35 fixes), and James (9 fixes). CWE-470 accounted for 9.99% (171) with significant contributions from Apache Hadoop (152 fixes) and Eclipse GlassFish (8 fixes). CWE-918 followed, representing 10.46% (179), mainly

in Eclipse GlassFish (121 fixes) and Apache CXF (23 fixes). Lastly, CWE-79 accounted for 7.77% (133), predominantly in Eclipse GlassFish (108).

ChatGPT 4 mitigated 767 vulnerabilities with CWE-23 appearing as the most frequently addressed issue with 27.64% (212) of fixes, primarily in Eclipse GlassFish and Apache CXF. CWE-319 accounted for 17.47% (134), while CWE-470 followed closely with 15.78% (121). CWE-918 and CWE-79 represented 12.91% (99) and 10.17% (78), respectively with these vulnerabilities predominantly identified in Eclipse GlassFish, Apache NiFi, and Apache Hadoop. Claude 3.5 Sonnet as displayed in Graph (c) addressed 717 vulnerabilities, exhibiting a distribution pattern similar to ChatGPT 4. CWE-23 was the most commonly fixed vulnerability, accounting for 27.62% (198), followed by CWE-319 at 17.30% (124) and CWE-470 at 15.35% (110). CWE-918 and CWE-79 accounted for 12.83% (92) and 10.18% (73), respectively with a notable concentration of these issues in Eclipse GlassFish, Apache NiFi, and Apache CXF. Gemini 2.0 Flash performed comparably, resolving 688 vulnerabilities in total. CWE-23 remained the most frequently fixed vulnerability, constituting 27.33% (188), followed by CWE-319 at 16.72% (115) and CWE-470 at 15.70% (108). CWE-918 and CWE-79 accounted for 12.94% (89) and 10.03% (69), respectively with fixes mostly in Eclipse GlassFish, Apache NiFi, and Apache Hadoop. Llama 3.2 (3B) addressed 620 out of 1,897 vulnerabilities with CWE-23 most frequent at 24.52% (152), concentrated in Eclipse GlassFish, Apache NiFi, and Apache CXF. CWE-319 followed at 16.94% (105), mainly in Eclipse GlassFish and Apache NiFi, while CWE-470 accounted for 15.32% (95) in Apache Hadoop, and CWE-918 and CWE-79 for 12.58% (78) and 8.71% (54), respectively, in Eclipse GlassFish and Apache CXF.

Across all models, Eclipse GlassFish and Apache Hadoop consistently exhibited the highest concentrations of fixed vulnerabilities, underscoring the urgent need for targeted security enhancements in these codebases. The varying performance levels among

the models highlight the distinct approaches each takes toward vulnerability mitigation with VulnFixAI leading in overall fixes, followed closely by ChatGPT 4, Claude 3.5 Sonnet, and Gemini 2.0 Flash, while Llama 3.2 (3B) showed more limited coverage.

These findings highlight the critical role of leveraging VulnFixAI's hybrid approach vulnerability mitigation strategies to enhance software security across diverse open-source projects, offering valuable insights for prioritizing security efforts in high-risk codebases.

#### 5.0.2.3 Analysis of Unfixed Vulnerabilities

While VulnFixAI achieved high effectiveness, a subset of identified vulnerabilities across the evaluated projects remained unfixed. These vulnerabilities, totaling 218 instances across 16 of the 20 evaluated projects, were only located in JUnit test code. Out of 2,362 total vulnerabilities identified across the projects, these unfixed instances represent approximately 11%, underscoring that the vast majority 89% were fixed. These instances were excluded from repair, as VulnFixAI is designed to prioritize vulnerabilities in production software where security risks are real, rather than in test code, which is executed only during development or testing phases and does not impact deployed systems. This method choice aligns with the tool's focus on delivering actionable fixes for real-world deployment, avoiding unnecessary refactoring in non-production contexts.

Table 5.1 summarizes the distribution of unfixed vulnerabilities by project. Apache Hadoop exhibited the highest count 54 instances, mostly in LOIS 30 and LOWV 24, reflecting extensive test code usage in this distributed system. Apache Flink followed with 38 instances with a notable concentration in LOWV 19 due to CWE-23 vulnerabilities in test cases. Other projects, such as FitNesse 1 instance and eclipse-vertx 1 instance, showed minimal unfixed vulnerabilities, suggesting less reliance on vulnerable test code.

This variability indicates that unfixed instances correlate with project complexity and test suite size, though they remain a small fraction of the total vulnerabilities evaluated.

Table 5.1
Unfixed Vulnerabilities by Project

| Project Name | ITV | LOWV | LOIS | Total Unfixed |
|---|---|---|---|---|
| Apache CXF | 8 | 8 | 2 | 18 |
| Apache Dubbo | 2 | 0 | 2 | 4 |
| Apache Flink | 18 | 19 | 1 | 38 |
| Apache Hadoop | 0 | 24 | 30 | 54 |
| Apache Hive | 10 | 7 | 12 | 29 |
| Apache NiFi | 8 | 4 | 2 | 14 |
| Eclipse GlassFish | 0 | 0 | 4 | 4 |
| FitNesse | 0 | 0 | 1 | 1 |
| Hugegraph-toolchain | 1 | 0 | 2 | 3 |
| Infinispan | 4 | 1 | 4 | 9 |
| James | 6 | 0 | 0 | 6 |
| Keycloak | 2 | 2 | 1 | 5 |
| eclipse-vertx | 0 | 0 | 1 | 1 |
| netty | 6 | 0 | 5 | 11 |
| openmeetings | 2 | 0 | 1 | 3 |
| undertow | 14 | 1 | 3 | 18 |

Table 5.2 shows the unfixed vulnerabilities by category. LOIS accounted for the largest percentage 83 instances, 38.1%. ITV followed with 80 instances 36.7%. LOWV had 55 instances 25.2%. The distribution of LOIS and ITV vulnerabilities across test code appears more frequent than LOWV, due to the nature of test scenarios that explore input handling and trust-related edge cases more extensively than whitelist validation.

Further analysis by CWE-ID shows different patterns. CWE-23 (Path Traversal) was the most frequent with 56 instances 25.7% of unfixed vulnerabilities, concentrated heavily in Apache Flink 19 and Apache Hadoop 24, reflecting its common use in testing file system interactions. Conversely, several CWE-IDs tied for the lowest count at 1 instance each, including CWE-89 (SQL Injection) in Apache Flink, CWE-78 (OS Command Injection) in FitNesse, and CWE-547 (Hardcoded Crypto Keys) in Infinispan, indi-

Table 5.2
Unfixed Vulnerabilities by Category

| Vulnerability Category | Count | Percentage | CWE IDs |
|---|---|---|---|
| ITV | 80 | 36.70% | CWE-295,CWE-296, CWE-275, CWE-297, CWE-298, CWE-299, CWE-599, CWE-319,CWE-320, CWE-321, CWE-326, CWE-327,CWE-330, CWE-329, CWE-353, CWE-345, CWE-346, CWE-347, CWE-358,CWE-775, CWE-401, CWE-494, CWE-502, CWE-522, CWE-532, CWE-547, CWE-668, CWE-732, CWE-916, CWE-918, CWE-940, CWE-941 CWE-1021. |
| LOWV | 55 | 25.20% | CWE-20, CWE-23, CWE-36, CWE-41, CWE-59, CWE-74, CWE-77, CWE-119, CWE-120, CWE-130, CWE-131, CWE-134,CWE-179, CWE-180, CWE-181, CWE-182, CWE-183, CWE-184, CWE-185, CWE-186,CWE-208, CWE-306, CWE-354, CWE-434, CWE-444, CWE-470, CWE-501, CWE-601, CWE-611, CWE-641 CWE-707, CWE-787, CWE-943, CWE-1173, CWE-1919. |
| LOIS | 83 | 38.10% | CWE-74, CWE-75, CWE-76,CWE-77, CWE-78, CWE-79, CWE-80, CWE-83, CWE-85, CWE-86, CWE-88, CWE-89, CWE-90, CWE-91, CWE-93, CWE-94, CWE-95, CWE-96, CWE-97,CWE-98, CWE-99, CWE-113, CWE-116, CWE-117, CWE-134, CWE-138, CWE-140, CWE-150, CWE-151, CWE-152, CWE-157, CWE-158, CWE-159, CWE-160, CWE-165, CWE-166, CWE-167,CWE-642, CWE-176,CWE-352, CWE-434, CWE-470, CWE-471, CWE-502, CWE-564, CWE-601, CWE-614, CWE-643, CWE-917. |

cating their rarity in test code across the dataset. CWE-23's dominance underscores its relevance in test suites for projects handling file operations, while the sparse occurrence of other CWE-IDs suggests they are less critical or less tested in non-production contexts.

The exclusion of these 218 test-code vulnerabilities contributes to the 11%–12% of unfixed cases in each category (e.g., 89% fix rate for ITV and LOWV, 88% for LOIS). Given the dataset's focus on production-relevant fixes, this trade-off strengthens VulnFixAI's useful. The exclusion of these 218 test-code vulnerabilities accounts for 11%–12% of unfixed cases in each category (e.g., 89% fix rate for ITV and LOWV, 88% for LOIS). Since VulnFixAI prioritizes production-suitable fixes, this trade-off enhances its practical applicability by focusing on real-world software vulnerabilities rather than test-related artifacts.

#### 5.0.2.4 Strengths of VulnFixAI Across Vulnerability Types

This section presents four real-world Java vulnerability examples to demonstrate how VulnFixAI successfully detects and mitigates vulnerabilities that general-purpose LLMs fail to resolve.

#### 5.0.2.5 Lack of Input Sanitization (LOIS)

This category includes vulnerabilities like XSS and HTTP Response Splitting, where unsafe user inputs are directly incorporated into output without proper sanitization. VulnFixAI uses OSR to mitigate these risks by applying encoding and sanitization libraries.

As shown in Listing 5.7, the vulnerability arises from the direct flow of user-controlled input into a method that generates part of the application's HTTP response, without applying proper output encoding. This weakness creates an opportunity for reflected cross-site scripting.

```
1 public String genReportByAuto(@Validated @RequestBody AutoReportRequest request) {
2   return testPlanReportService.genReportByAuto(request); // <———CWE—79
3 }
```

Listing 5.7: CWE-79 – Cross-Site Scripting (XSS)

The exact vulnerable line is `return testPlanReportService.genReportByAuto(request)`, the request object, populated from user input but is passed without sanitization to $genReportByAuto$, which eventually renders output for the client. If malicious payloads are included in the input, they can be injected into the response HTML, enabling XSS. The absence of output encoding (e.g., HTML escaping) means the browser interprets the injected scripts as executable content.

General-purpose LLMs such as ChatGPT-4 and Claude 3.5 did not recognize the end-to-end flow from request input to response rendering. They incorrectly assumed that $@Validated$ ensures safety, and suggested minimal changes such as try-catch blocks

or input type validation. They failed to recommend proper output encoding, such as using $Encode.forHtml(...)$, which is critical to mitigating CWE-79. VulnFixAI, informed by domain-specific CWE patterns, flagged this flow as unsafe and applied output sanitization using OWASP Java Encoder.

As shown in Listing 5.8, the vulnerability arises from inserting user-controlled input directly into an HTTP header without proper validation. This can allow attackers to inject carriage return and line feed characters $(nrnn)$, resulting in HTTP response splitting and potential header injection attacks.

```
1 public void testDelegateThenAuthenticate() throws IOException {
2   HttpHeaders optionsBuilder = new HttpHeaders();
3   optionsBuilder.addHeader("Authorization", "Bearer " + token); // <———CWE—113
4   HttpResponse response = httpClient.sendRequest(optionsBuilder);
5 }
```

Listing 5.8: CWE-113 – HTTP Response Splitting

The exact vulnerable line is `optionsBuilder.addHeader("Authorization", "Bearer " + token);`, the token is supposed to be retrieved from a remote or user-influenced source. If an attacker manages to include newline characters in the token, it could terminate the header prematurely and inject additional headers or even response bodies. This results in HTTP response splitting CWE-113, which can be exploited for cross-site scripting, cache poisoning, or redirect attacks.

General-purpose LLMs such as ChatGPT-4 and Claude 3.5 failed to recognize the implications of control characters in HTTP headers. They focused on input presence checks (e.g., $null/emptyvalidations$) and did not recommend filtering out $CR(nr)$ and $LF(nn)$ characters. VulnFixAI, by contrast, leveraged its CWE-113 training and enforced TER by introducing explicit checks against these characters and rejecting malformed tokens before they are inserted into headers. This approach prevents header manipulation and ensures compliance with secure HTTP construction practices.

As shown in Listing 5.9, the vulnerability arises from dynamically constructing a file system path using values derived from potentially untrusted environment sources and passing it directly to a method that may invoke OS-level commands. This can lead to arbitrary command execution if not properly validated.

```
1  public static void showLog() {
2      if (RevealFileAction.isSupported()) {
3          RevealFileAction.openDirectory(PathManager.getHomePath() + "/logs"); // <———CWE—78
4      }
5  }
```

Listing 5.9: CWE-78 – OS Command Injection

The exact vulnerable line is `RevealFileAction.openDirectory(PathManager.getHomePath()+ "/logs");` The method constructs a file path using the value returned by $PathManager.getHomePath()$, which could be influenced by environment variables or external configuration. This path is then passed to $openDirectory(...)$, which may delegate to a native system call or shell command. If the path contains malicious input, such as command injection payloads (e.g., $;rm-rf/$), it could result in the execution of unintended commands on the host system.

General-purpose LLMs treated the line as a simple file operation and overlooked the security implications of passing unchecked strings to system-level functions. Their recommendations were limited to adding null checks or try-catch blocks, without recognizing the need to enforce path constraints. In contrast, VulnFixAI identified the potential injection point and applied WVR. It introduced path normalization using `Paths.get(...).normalize()` and enforced a base directory constraint, effectively preventing directory traversal and command injection.

#### 5.0.2.6 Lack of Whitelist Validation (LOWV))

This category targets vulnerabilities arising from accepting unchecked inputs, such as filenames, commands, or query strings, without enforcing input conformity. Vuln-

FixAI uses Whitelist Validation Refactoring (WVR) to enforce regular expression-based validation against strict input constraints.

As shown in Listing 5.10, the vulnerability arises from writing user-controlled input directly into a logging method that supports format specifiers. Without proper sanitization, this can lead to unexpected behavior, memory access issues, or denial-of-service, depending on how the format string is processed.

```
1  public void messageReceived(final ChannelHandlerContext ctx, final MessageEvent e) {
2    String request = (String) e.getMessage();
3    Logger.info(request); } // <———CWE—134
```

Listing 5.10: CWE-134 – Uncontrolled Format String

The exact vulnerable line is $Logger.info(request)$;, the request originates from client input (e.g., cookies, HTTP body) and is passed directly to the logger as the format string. If the input contains special format specifiers such as $\%s$, $\%n$, or $\%x$, the logger may interpret them as instructions instead of plain text. This could lead to corrupted logs or exposure of internal data structures, especially in loosely configured logging environments.

General-purpose LLMs failed to identify the string as a format string. They suggested surrounding the logging call with a try-catch block or recommended logging input as-is. These responses did not eliminate the root cause. VulnFixAI recognizes CWE-134, and applies TER by transforming the logging pattern to use parameterized logging.

As shown in Listing 5.11, the vulnerability arises from the use of externally-controlled input in a reflection context to dynamically load classes. If the class name is not properly validated, an attacker can instantiate arbitrary classes or execute malicious code.

```
1  private void mkdir(FileContext fc, Path path, FsPermission permission) throws IOException {
2    FileStatus fsStatus = fc.getFileStatus(path); // <———CWE—470
3    if (fsStatus != null) { return; }
4    fc.mkdir(path, permission, true); }
```

Listing 5.11: CWE-470 – Unsafe Reflection

The exact vulnerable line is $FileStatus fsStatus = fc.getFileStatus(path);$, the path value may originate from untrusted input, such as configuration files or environment variables. When passed to $fc.getFileStatus(path)$, it may invoke underlying class resolution or load mechanisms depending on the file system abstraction in use. Without sanitizing or restricting the type or origin of the path, this allows attackers to influence file system behavior or trigger reflection in plugin-based contexts.

General-purpose LLMs like ChatGPT-4 and Gemini failed to associate `getFileStatus(path)` with potential unsafe reflection risks. They treated the call as a harmless I/O check. None of the models suggested enforcing a whitelist of valid path schemes (e.g., $hdfs://$, $file://$) or filtering input source. VulnFixAI, informed by CWE-470 patterns, flagged the risk and applied WVR by constraining input origins and introducing type checks on the resolved FileSystem instance to prevent dynamic loading of unintended file system handlers.

As shown in Listing 5.12, the vulnerability arises from constructing a file path using externally-controlled input without validating or normalizing the path, allowing attackers to traverse directories and access unauthorized files.

```
1 private static URL checkConfigFile(File f) {
2   return (f.exists() && f.isFile()) ? f.toURI().toURL() : null; // <———CWE—23
3 }
```

Listing 5.12: CWE-23 – Relative Path Traversal

The exact vulnerable line is$return(f.exists()\&\&f.isFile())?f.toURI().toURL() : null$;, the method accepts a File object, potentially created from a path supplied by an external source and converts it directly into a URL without performing path sanitization. If the original path includes relative segments (e.g., ../../), the check may pass and allow access to sensitive configuration files outside the intended directory scope, resulting in a classic path traversal attack (CWE-23).

ChatGPT-4 and Claude 3.5 treated this method as a safe configuration check and focused on whether the file exists or is null. None of them recognized that the method could allow directory escape through crafted paths. Furthermore, they did not recommend applying canonicalization or restricting the path to a safe base directory. VulnFixAI applied WVR by normalizing the path with $getCanonicalPath()$ and ensuring it remained within a predefined safe root directory, thereby eliminating the path traversal vector.

#### 5.0.2.7 Inadequate TrustChain Verification (ITV)

This category addresses vulnerabilities arising from weak validation of external inputs or trust boundaries, such as remote URLs or deserialization. VulnFixAI applies TrustChain Verification Refactoring (TCVR) to enforce secure verification steps such as certificate validation or input source whitelisting.

As shown in Listing 5.13, the vulnerability arises when untrusted input is used to construct outbound requests without validating the target address. This enables attackers to abuse server privileges to access internal services or unauthorized resources.

```
1 public HttpURLConnection run() throws Exception {
2   return authUrl.openConnection(url, authToken, doOutput); // <———CWE—918
3 }
```

Listing 5.13: CWE-918 – Server-Side Request Forgery (SSRF)

The exact vulnerable line is $returnauthUrl.openConnection(url, authToken, doOutput)$, the url value is assumed to be externally controlled, possibly from a configuration, HTTP parameter, or chained resource. If not validated, an attacker can supply a crafted internal endpoint (e.g., $http://localhost:8080/admin$) to access restricted services or trigger unintended internal requests. This results in CWE-918, allowing exploitation of trusted network positions through forged server-side requests.

General-purpose LLMs such as Claude 3.5 and Gemini 2.0 either ignored the dynamic nature of the url or suggested generic exception handling for $openConnection(...)$.

They did not recognize the request as originating from potentially malicious user input, nor did they recommend validating destination hosts or protocols. VulnFixAI applied TCVR. It added logic to restrict requests to an approved domain allowlist and verified that only permitted protocols like HTTPS were used, thereby blocking access to sensitive internal services.

```
1 private void invoke() throws Exception {
2   InputStream is = sock.getInputStream(); // <———CWE—319
3   // ... processing input
4 }
```

Listing 5.14: CWE-319 – Cleartext Transmission of Sensitive Information

The exact vulnerable line is $InputStream is = sock.getInputStream();$. In this instance, the application reads from a raw socket $sock$ without verifying whether the connection is secured using $SSL/TLS$. If this socket operates over a plain-text protocol (e.g., HTTP or an unencrypted TCP stream), sensitive information, including authentication headers, session cookies, or payload data, may be transmitted in cleartext.

General-purpose LLMs like ChatGPT-4 and Gemini viewed this line as harmless I/O initialization. They failed to inquire about or enforce the use of secure socket layers. Some suggested wrapping the stream in a buffer or logging exceptions but did not consider transport-layer security. In contrast, VulnFixAI identified this as a cleartext transmission risk and applied TCVR by enforcing that sockets be instances of SSLSocket or wrapping the communication in an SSLContext. It inserted verification checks to ensure encryption before data transmission began.

# CHAPTER SIX

# DISCUSSION

This section addresses the research questions posed in Section 1, drawing on the findings presented in the preceding sections.

***RQ1****: What are the different types of software vulnerabilities categorized by their root causes?* Based on a meticulous analysis of the root causes and common traits of CWE types (cf. Table 3.1), this work has defined eight categories of software vulnerabilities, each with severity level, refining the general categories of the CWE system. They include Lack of Boundary Checks (LOBC), Improper Array Copying (IAC), Lack of Input Sanitization (LOIS), Improper Numeric Handling (INH), Lack of Whitelist Validation (LOWV), Improper Type Handling (ITH), Insecure Error Feedback (IER), and Inadequate TrustChain Verification (ITV). These categories are finely delineated to encapsulate common characteristics within each group, thereby facilitating the development of solutions tailored to address the specific vulnerabilities within each category.

***RQ2****: What strategies should be employed to effectively address the vulnerabilities of these categories?* For each category, a specific refactoring technique has been developed to address the root causes associated with the vulnerabilities (cf. Table 3.2) of the category. These techniques include Boundary Sentinel Refactoring (BSR) addressing LOBC, Safe Array Copy Refactoring (SACR) addressing IAC, Output Safety Refactoring (OSR) addressing LOIS, Numeric Safety Refactoring (NSR) addressing INH, Whitelist Validation Refactoring (WVR) addressing LOWV, Type Enforcement Refactoring (TER) addressing ITH, Input Feedback Refactoring (IFR) addressing IEF, and TrustChain Verification Refactoring (TCVR) addressing ITV. Each technique is tailored to address the

root causes within its corresponding category, yet remains broadly applicable to various vulnerability types in that category.

***RQ3**: Is there any interdependence among the proposed refactoring techniques?* There exist interdependencies among the proposed refactoring techniques (cf. Figure 5.5), which can complement and support each other, thereby strengthening the overall security of systems. SACR can be enhanced by WVR to ensure only pre-approved data sizes are processed. BSR may leverage WVR to pre-screen data for safety criteria, ensuring robust buffer management. OSR and IFR can be used together to improve the clarity and safety of feedback provided to users, thus enhancing secure user interactions. TER and NSR can be jointly used to prevent type-related errors and numeric overflows. WVR and IFR together can improve data security and user guidance.

***RQ4**: How effective are the proposed refactoring techniques in addressing targeted vulnerabilities?* The evaluation conducted on twenty-one real-world projects results in the overall reduction of 88% (reduced to 335 from 2,879) by the proposed refactoring techniques (cf. Figure 5.3 and Figure 5.4), demonstrating their effectiveness. The most reduction was observed in Apache-OFBiz and CoreNLP at 97%, while the least reduction was shown in OpenRemote at 52%. In terms of domain, the AI domain shows the most reduction by 96%, while the least reduction was observed in IoT domain at 62%. The remaining vulnerabilities (11%, 335 out of 2,879) were related to JUnit tests and dependencies with third-party libraries. The overall substantial decrease across diverse projects and domains confirms the effectiveness of the proposed refactoring techniques in addressing vulenerabilities.

It is notable that the proposed approach is not limited to trivial cases but also demonstrates applicability to complex real-world vulnerabilities. The case study on CVE-2021-44228 in Section 3.0.1.11 illustrates how multiple refactoring techniques can be applied in combination to address a critical vulnerability spanning several categories,

including LOWV, and ITH. By integrating WVR and TER, the underlying issues were effectively mitigated, preventing unsafe lookups, blocking unauthorized endpoints, and enforcing strict type constraints. This supports that the proposed refactoring techniques can be systematically applied to address security risks in real-world scenarios.

***RQ5**: To what extent does fine-tuning DSLLMs on a domain-specific dataset improve the accuracy of vulnerability detection and automated fixes compared to non-fine-tuned DSLLMs?* Fine-tuning DSLLMs significantly improves vulnerability identification accuracy across all three categories: 100% for LOWV, ITV, and LOIS, compared to 72%, 76%, and 85% for the non-fine-tuned version(cf. Figure 5.6). This demonstrates a 24–28% improvement in detection capability, reinforcing the importance of training AI models on domain-specific datasets. Moreover, The fine-tuned DSLLMs achieved fix rates of 89% for LOWV and ITV, and 88% for LOIS, whereas the non-fine-tuned achieved only 75%, 77%, and 69%, respectively(cf. Figure 5.7). These results confirm that fine-tuning not only enhances detection but also enables more accurate, structured fixes. VulnFixAI demonstrated higher consistency in security-related fixes, highlighting the benefit of domain-specific model training.

***RQ6**: How can a hybrid approach combining fine-tuned DSLLMs and refactoring algorithms enhance the automated detection and repair of software vulnerabilities compared to existing state-of-the-art models?*

VulnFixAI's hybrid approach combining fine-tuned Llama 3.2 (3B) model with WVR, OSR, and TCVR delivers marked improvements in detecting and repairing vulnerabilities compared to state-of-the-art models. VulnFixAI achieved an overall effectiveness rate of 89%(cf. Figure 5.8) with a 100% identification rate and fix rates of 85-89%(cf. Figure 5.6, and cf. Figure 5.7). In contrast, leading LLMs) ike ChatGPT-4, Claude 3.5 Sonnet, and Gemini 2.0 Flash recorded effectiveness rates of 71-74% with fix rates peaking at 84-86%(cf. Figure 5.6 and cf. Figure 5.7). This represents a 15-18% improvement

over these models and a 51% leap over the non-fine-tuned Llama 3.2 (3B), which scored 59%.

***RQ7**: How do fine-tuned DSLLMs perform across different vulnerability categories (e.g., LOWV, LOIS, ITV) in terms of repair consistency, security robustness, and generalization compared to LLMs?*

The evaluation results in Section 5.0.2.1 and 5.0.2.2 reveal that fine-tuned DSLLMs, as implemented in VulnFixAI, demonstrate significantly better performance across all three vulnerability categories LOWV, LOIS, and ITV compared to general-purpose LLMs.

VulnFixAI's performance across the LOWV, ITV, and LOIS categories is consistently strong, though influenced by category-specific factors. It achieves a 100% identification rate across all three with fix rates of 89% LOWV, 89% ITV, and 88% LOIS(cf. Figure 5.6, and cf. Figure 5.7), translating to overall effectiveness rates of 88-89%. Compared to existing LLMs at 59-77% effectiveness(cf. Figure 5.8.) VulnFixAI excels in addressing these diverse weaknesses.

These improvements are attributed to the integration of domain-specific refactoring techniques (WVR, OSR, TCVR) and fine-tuned DSLLMs, which enable precise, context-aware repairs. The results also demonstrate VulnFixAI's repair consistency, as the same refactoring patterns were correctly and applied across diverse projects and CWE types, such as CWE-23, CWE-79, and CWE-319, without introducing regressions or inconsistencies.

VulnFixAI's hybrid approach enables not just detection but effective mitigation of complex security weaknesses across multiple CWEs under each category. By implanting refactoring repair logic into the model and training on 10,000 curated vulnerable Java snippets from 907 projects, VulnFixAI could generalize its repair capability to real-world vulnerabilities. Notably, the model succeeded in repairing high-severity vulnera-

bilities such as CWE-470 and CWE-918, which were often mishandled by other LLMs. In contrast, general-purpose LLMs struggled to generate reliable fixes, particularly for less common or structurally complex vulnerabilities, due to their reliance on broad, non-specialized training data.

These findings confirm that fine-tuned DSLLMs offer a robust and scalable solution for vulnerability detection and automated repair when enhanced by security-specific design.

The findings of this work have significant practical implications for software developers responsible for maintaining secure codebases. By introducing a set of refactoring techniques specifically designed to mitigate vulnerabilities, this research provides developers with actionable strategies that go beyond generic code improvement. The integration of these techniques into VulnFixAI enables automated detection and repair of vulnerabilities in Java code, decreasing the dependence on manual code reviews and lowering the barrier to applying secure development practices. This not only accelerates the vulnerability mitigation process but also ensures more consistency and precision in the applied fixes. Furthermore, the use of fine-tuned DSLLMs allows VulnFixAI to produce context-aware patches that align with real-world security requirements, making it a practical and scalable tool for use in continuous integration pipelines and secure software development lifecycles.

## 6.1 Threats to Validity

This section discusses potential threats to the validity of this study and the measures taken to mitigate them.

**Internal Validity:** One potential threat to internal validity is the manual validation of vulnerability samples generated by ChatGPT. Although all 405 generated samples were carefully reviewed to ensure correctness, human biases and errors may have influ-

enced the process. To mitigate this, the samples were cross-checked against the CWE descriptions and validated them based on known vulnerability patterns. However, using ChatGPT for vulnerability generation may introduce data leakage or biases from its training corpus, potentially affecting the realism of the generated samples. Future work could involve augmenting the dataset with real-world vulnerabilities.

Although each of the 405 generated vulnerability samples was manually validated by domain experts to ensure alignment with CWE definitions, the process is still subject to human error. Potentially impacting the consistency and reliability of the dataset used for designing refactoring techniques.

Another internal validity threat stems from the dataset used to fine-tune the Llama 3.2 (3B) model. Although 10,000 vulnerable Java snippets were curated from 907 open-source projects, only 1,000 (10%) were manually validated for accuracy.This limited validation introduces a risk of undetected errors, such as misclassified vulnerabilities or incomplete fixes, in the remaining data. To address this, CodeQL static analysis was applied to verify refactored snippets, ensuring fix quality, though minor inaccuracies may persist. However, reliance on static analysis introduces its own limitations, CodeQL may produce false positives or overlook context-specific vulnerabilities, and the effectiveness of validation depends on the quality of the queries used. Thus, minor inaccuracies may still persist in both labeling and fix verification. Another threat arises from comparing VulnFixAI to baseline LLMs (e.g., ChatGPT-4, Claude 3.5 Sonnet). These models, lacking domain-specific fine-tuning, may have been disadvantaged, potentially exaggerating VulnFixAI's improvement. This threat was mitigated by using identical Alpaca-formatted inputs across all models, but differences in training data and architecture remain potential confounders.

**External Validity:** This study focused on vulnerabilities in Java applications. While the refactoring principles may be applicable to other programming languages, the

specific implementations of refactoring techniques may vary due to language-specific constructs and security risks. Future studies should investigate whether similar refactoring strategies can be applied effectively to other languages such as Python, C++, or JavaScript.

Additionally, the refactoring evaluation in this work is based on vulnerability reduction rather than overall software quality. While this results demonstrate an 89% reduction in vulnerabilities, the impact of refactoring on other software quality attributes, such as maintainability, performance, and scalability, was not explicitly measured. Some refactoring techniques may introduce additional overhead, affect code readability, or require developer training for proper implementation. Future studies should consider the trade-offs between security improvement and software maintainability to provide a more holistic evaluation.

Another key limitation is VulnFixAI's focus on Java code, which restricts its applicability to other languages like C, C++, or Python. The dataset, sourced from 907 open-source projects in GitHub, may not reflect the security challenges of private or closed-source Java systems, potentially limiting effectiveness in those domains. Furthermore, the evaluation on 20 real-world projects, while diverse in size (66 to 16,850 lines) and domain (e.g., Android, web applications), may not capture the full range of Java applications, such as those with unique development practices or complexity, For instance, projects utilizing weird coding styles e.g., heavy reliance on reflection, extensive use of third-party frameworks with custom security configurations, or domain-specific libraries might introduce vulnerability patterns not well-represented in the dataset. Similarly, highly complex systems, such as large-scale enterprise applications with complex interdependencies, multi-threaded architectures, or legacy codebases maintained over decades, could challenge VulnFixAI's detection and repair capabilities due to increased context sensitivity or refactoring constraints. To enhance generalizability, projects with

confirmed CVEs and active maintenance were selected, but more comprehensive testing across additional contexts is needed to confirm VulnFixAI's robustness.

Additionally, while the proposed refactoring techniques were evaluated on Java projects ranging from Java 8 to Java 17, compatibility with all Java versions is not guaranteed. Language features introduced in versions beyond Java 17 may not be fully supported or accounted for in the current implementation. As a result, the effectiveness and applicability of the techniques in projects using newer Java versions could be limited, representing a potential threat to external validity.

# CHAPTER SEVEN

# CONCLUSION AND FUTURE WORK

This dissertation has introduced a refactoring approach to address software vulnerabilities. It defines eight fine-grained categories of vulnerabilities based on their root causes, each associated with severity level. Tailored refactoring solutions are developed for each category to address its unique vulnerabilities. The proposed refactoring techniques are interdependent, complementing each other to strengthen the overall security of systems. Their effectiveness was evaluated through twenty-one real-world applications, demonstrating an average reduction of 88% in high-severity vulnerabilities, confirming their value alongside traditional detection and patching methods.

Building on this foundation, VulnFixAI was developed as a novel tool that synergistically integrates fine-tuned DSLLMs, Llama 3.2 (3B parameters) with three specialized refactoring WVR, OSR, andbTCVR to automate the detection and repair of vulnerabilities in Java code. This approach tackles the limitations of traditional static and dynamic analysis tools as well as general-purpose large language models (LLMs) by offering a hybrid methodology that combines precise vulnerability identification with systematic, security-focused remediation. Evaluated on 20 real-world projects from the GitHub Advisory Database, VulnFixAI achieved an overall effectiveness rate of 89% with a 100% identification rate and fix rates of 88–89% across LOWV, ITV, and LOIS categories. These results reflect a 15–18% improvement over leading large language models (LLMs) like ChatGPT-4 (71–74%) and a 51% jump over non-fine-tuned Llama 3.2 (59%), emphasizing the power of combining domain-specific fine-tuning with systematic refactoring. Unlike traditional static analysis tools or probabilistic LLM outputs, VulnFixAI delivers scalable, security-centric repairs validated by CodeQL, addressing a critical gap

in software vulnerability. Future work will include increasing the dataset to enhance coverage, and adding support for additional programming languages such as C, C++, and Python, and incorporating systematic dependency analysis to better capture the relationships among program elements. This will enable more context-aware refactoring, improve prioritization of security fixes based on code interdependencies, and support compound mitigation strategies for interrelated vulnerabilities.

# REFERENCES


[1] Chaima Abid, Marouane Kessentini, Vahid Alizadeh, Mouna Dhaouadi, and Rick Kazman. How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, 48(3):864–878, 2020.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[3] Ehsan Aghaei and E. Al-Shaer. Threatzoom: neural network for automated vulnerability mitigation. *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, 2019.

[4] Georgios Aivatoglou, Mike Anastasiadis, Georgios Spanos, A. Voulgaridis, K. Votis, and D. Tzovaras. A tree-based machine learning methodology to automatically classify software vulnerabilities. *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 312–317, 2021.

[5] Abdullah Almogahed, Mazni Omar, and Nur Haryani Zakaria. Refactoring codes to improve software security requirements. *Procedia Computer Science*, 204:108–115, 2022.

[6] Bandar Alshammari, Colin Fidge, and Diane Corney. Assessing the impact of refactoring on security-critical object-oriented designs. In *Asia Pacific Software Engineering Conference*, pages 186–195, 2010.

[7] anthropic. claude. Available online, 2023.

[8] Apache Log4j. Log4j GitHub Repository. https://github.com/apache/logging-log4j2. Accessed: 2025-03-25.

[9] Apache OpenMeetings. Apache OpenMeetings GitHub Repository. https://github.com/apache/openmeetings. Accessed: 2024-04-9.

[10] Apache Software Foundation. Apache Maven. https://maven.apache.org. Accessed: 2024-05-25.

[11] Syafiq Al Atiiq, Christian Gehrmann, Kevin Dahl'en, and Karim Khalil. From generalist to specialist: Exploring cwe-specific vulnerability detection. *ArXiv*, abs/2408.02329.

[12] Andrew Austin, Casper Holmgreen, and Laurie Williams. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology*, 55(7):1279–1288, 2013.

[13] Pavel Avgustinov, Kevin Backhouse, and Man Yue Mo. Variant analysis with ql. In *Formal Methods: 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings 22*, pages 666–670, 2018.

[14] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

[15] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 19:1617–1664, 2014.

[16] Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. Deepcode ai fix: Fixing security vulnerabilities with large language models. *arXiv preprint arXiv:2402.13291*, 2024.

[17] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[18] Bitcoinj. bitcoinj GitHub Repository. https://github.com/bitcoinj/bitcoinj. Accessed: 2024-08-1.

[19] BroadleafCommerce. BroadleafCommerce GitHub Repository. https://github.com/BroadleafCommerce/BroadleafCommerce. Accessed: 2024-04-9.

[20] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468, 2022.

[21] Sicong Cao, Xiaobing Sun, Ratnadira Widyasari, David Lo, Xiaoxue Wu, Lili Bo, Jiale Zhang, Bin Li, Wei Liu, Di Wu, et al. A systematic literature review on explainability for machine/deep learning-based software engineering research. *arXiv preprint arXiv:2401.14617*, 2024.

[22] Jiaqi Chang, Zhujuan Ma, Binghao Cao, and Erzhou Zhu. Vdda: An effective software vulnerability detection model based on deep learning and attention mechanism. In *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 474–479, 2023.

[23] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[24] Xiaoyi Chen, Siyuan Tang, Rui Zhu, Shijun Yan, Lei Jin, Zihao Wang, Liya Su, Zhikun Zhang, XiaoFeng Wang, and Haixu Tang. The janus interface: How fine-tuning in large language models amplifies the privacy risks. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1299, 2024.

[25] Yupan Chen and Zhihong Liu. Hlt: A hierarchical vulnerability detection model based on transformer. In *2022 4th International Conference on Data Intelligence and Security (ICDIS)*, pages 50–54, 2022.

[26] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.

[27] JBoss Community. Undertow. https://github.com/undertow-io/undertow. Accessed: 2025-03-21.

[28] CoreNLP. kura GitHub Repository. https://github.com/stanfordnlp/CoreNLP. Accessed: 2024-08-1.

[29] MITRE Corporation. Common weakness enumeration (cwe). https://cwe.mitre.org. Accessed: 2024-02-5.

[30] Yiming Cui, Ziqing Yang, and Xin Yao. Efficient and effective text encoding for chinese llama and alpaca. *arXiv preprint arXiv:2304.08177*, 2023.

[31] Michael A Cusumano. Who is liable for bugs and security flaws in software? *Communications of the ACM*, 47(3):25–27, 2004.

[32] Apache CXF-Community. Cxf. https://github.com/apache/cxf. Accessed: 2025-03-21.

[33] Carlos Dantas, Adriano Rocha, and Marcelo Maia. Assessing the readability of chatgpt code snippet recommendations: A comparative study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, pages 283–292, 2023.

[34] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[35] Yukun Dong, Yeer Tang, Xiaotong Cheng, Yufei Yang, and Shuqi Wang. Sedsvd: Statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. *Information and Software Technology*, 158:107168, 2023.

[36] Kohei Dozono, T. Gasiba, and Andrea Stocco. Large language models for secure code assessment: A multi-language empirical study. *ArXiv*, abs/2408.06428, 2024.

[37] Apache Dubbo-Community. Apache dubbo. https://github.com/apache/dubbo. Accessed: 2025-03-21.

[38] Eclipse. Glassfish. https://github.com/eclipse-ee4j/glassfish. Accessed: 2025-03-21.

[39] Eclipse Kura. kura GitHub Repository. https://github.com/eclipse/kura. Accessed: 2024-08-1.

[40] Abdulrahman Abu Elkhail and Tomas Cerny. On relating code smells to security vulnerabilities. In *IEEE 5th intl conference on big data security on cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE intl conference on intelligent data and security (IDS)*, pages 7–12, 2019.

[41] Encog. kura GitHub Repository. https://github.com/jeffheaton/encog-java-core. Accessed: 2024-08-1.

[42] Fineract. fineract GitHub Repository. https://github.com/apache/fineract. Accessed: 2024-08-1.

[43] FitNesse. Fitnesse. https://github.com/unclebob/fitnesse. Accessed: 2025-03-21.

[44] Apache Flink-Community. Apache flink. https://github.com/apache/flink. Accessed: 2025-03-21.

[45] Apache Software Foundation. Apache ignite. https://github.com/apache/ignite. Accessed: 2024-03-26.

[46] Apache Software Foundation. Zookeeper github repository. https://github.com/apache/zookeeper?tab=readme-ov-file, 2024. Accessed: Accessed: 2024-03-25.

[47] Eclipse Foundation. Eclipse vert.x. https://github.com/eclipse-vertx/vert.x. Accessed: 2025-03-21.

[48] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[49] Michael Fu. Toward more effective deep learning-based automated software vulnerability prediction, classification, and repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 208–212, 2023.

[50] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. Vision transformer inspired automated vulnerability repair. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–29, 2024.

[51] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering*, 29(1):4, 2024.

[52] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 935–947, 2022.

[53] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 632–636, 2023.

[54] Shadi Ghaith and Mel Ó Cinnéide. Improving software security using search-based refactoring. In *International Symposium on Search Based Software Engineering*, pages 121–135, 2012.

[55] Github. Codeql. https://codeql.github.com. Accessed: 2025-03-30.

[56] GitHub. Github copilot: Your ai pair programmer. Available online, 2021.

[57] Apache Hadoop-Community. Apache hadoop. https://github.com/apache/hadoop. Accessed: 2025-03-21.

[58] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 9–pp, 2005.

[59] Jun Han and Yuliang Zheng. Security characterisation and integrity assurance for component-based software. In *Proceedings of International Conference on Software Methods and Tools*, pages 61–66, 2000.

[60] HAPI FHIR. HAPI FHIR GitHub Repository. https://github.com/hapifhir/hapi-fhir. Accessed: 2024-08-1.

[61] Red Hat. Infinispan. https://github.com/infinispan/infinispan. Accessed: 2025-03-21.

[62] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

[63] Apache Hive-Community. Apache hive. https://github.com/apache/hive. Accessed: 2025-03-21.

[64] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. Saver: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 271–283, 2020.

[65] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[66] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th international conference on software engineering*, pages 12–23, 2018.

[67] Zhen Huang and Marc White. Semantic-aware vulnerability detection. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 68–75, 2022.

[68] HugeGraph. Hugegraphtoolchain. https://github.com/hugegraph/hugegraph-toolchain. Accessed: 2025-03-21.

[69] HyperSQL. Hypersql db. https://github.com/hsqldb/hsqldb. Accessed: 2025-03-21.

[70] Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, and Fabio Palomba. Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security. *Empirical Software Engineering*, 28(4):89, 2023.

[71] Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea DeLucia, and Fabio Palomba. Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security. *Empirical Software Engineering*, 28(89):1–46, 2023.

[72] Ayano Ikegami, Raula Gaikovina Kula, Bodin Chinthanet, Vittunyuta Maeprasart, Ali Ouni, Takashi Ishio, and Kenichi Matsumoto. On the use of refactoring in security vulnerability fixes: An exploratory study on maven libraries. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pages 288–293, 2022.

[73] Nafis Tanveer Islam, Mohammad Bahrami Karkevandi, and Peyman Najafirad. Code security vulnerability repair using reinforcement learning with large language models. *arXiv preprint arXiv:2401.07031*, 2024.

[74] Apache James-Community. Apache james. https://github.com/apache/james-project. Accessed: 2025-03-21.

[75] JetBrains. IntelliJ IDEA. https://www.jetbrains.com/idea/. Accessed: 2024-05-25.

[76] Ossi Jormakka. Approaches and challenges of automatic vulnerability classification using natural language processing and machine learning techniques. 2019.

[77] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*, 2023.

[78] Katikapalli Subramanyam Kalyan. A survey of gpt-3 family large language models including chatgpt and gpt-4. *Natural Language Processing Journal*, 6:100048, 2024.

[79] T. Keefe. Software insecurity. https://www.computerworld.com/article/1328570/software-insecurity.html. Accessed: 2024-02-5.

[80] Keycloak. Keycloak. https://github.com/keycloak/keycloak. Accessed: 2025-03-21.

[81] Jingyue Li, Per Håkon Meland, Jakob Svennevik Notland, André Storhaug, and Jostein Hjortland Tysse. Evaluating the impact of chatgpt on exercises of a software security course. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2023.

[82] Junjie Li, Aseem Sangalay, Cheng Cheng, Yuan Tian, and Jinqiu Yang. Fine tuning large language model for secure code generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pages 86–90, 2024.

[83] Runhao Li, Chao Feng, Xing Zhang, and Chaojing Tang. A lightweight assisted vulnerability discovery method using deep neural networks. *IEEE Access*, 7:80079–80092, 2019.

[84] Yikun Li, Ting Zhang, Ratnadira Widyasari, Yan Naing Tun, Huu Hung Nguyen, Tan Bui, Ivana Clairine Irsan, Yiran Cheng, Xiang Lan, Han Wei Ang, et al. Cleanvul: Automatic function-level vulnerability detection in code commits using llm heuristics. *arXiv preprint arXiv:2411.17274*, 2024.

[85] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. Evaluating c/c++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[86] Bo Lin, Shangwen Wang, Liqian Chen, and Xiaoguang Mao. There are more fish in the sea: Automated vulnerability repair via binary templates. *arXiv preprint arXiv:2411.18088*, 2024.

[87] Elizabeth Lin, Igibek Koishybayev, Trevor Dunlap, William Enck, and Alexandros Kapravelos. Untrustide: Exploiting weaknesses in vs code extensions. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, 2024.

[88] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[89] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*, 2023.

[90] Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhai Wang. How chatgpt is solving vulnerability management problem. *arXiv preprint arXiv:2311.06530*, 2023.

[91] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology*, 33(5):1–26, 2024.

[92] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 2024.

[93] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 18–18, 2005.

[94] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 711–722, 2016.

[95] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22*, pages 229–246, 2017.

[96] Amel Mammar, Ana Cavalli, Willy Jimenez, Wissam Mallouli, and Edgardo Montes de Oca. Using testing techniques for vulnerability detection in c programs. In *Testing Software and Systems: 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings 23*, pages 80–96, 2011.

[97] Hazem W Marar. Advancements in software engineering using ai. *Computer Software and Media Applications*, 6(1):3906, 2024.

[98] Katsuhisa Maruyama and Takayuki Omori. A security-aware refactoring tool for java programs. In *Proceedings of the 4th Workshop on Refactoring Tools*, pages 22–28, 2011.

[99] P. Mell and A. Gueye. A suite of metrics for calculating the most significant security relevant software flaw types. In *Proceedings of the 44th Annual Computers, Software, and Applications Conference*, pages 511–516, 2020.

[100] Meta. React. https://react.dev. Accessed: 2025-03-23.

[101] Haris Mumtaz, Mohammad Alshayeb, Sajjad Mahmood, and Mahmood Niazi. An empirical study to improve software security through the application of code refactoring. *Information and Software Technology*, 96:112–125, 2018.

[102] Emanuele Antonio Napoli and Valentina Gatteschi. Evaluating chatgpt for smart contracts vulnerability correction. in 2023 ieee 47th annual computers. In *Software, and Applications Conference (COMPSAC)*, pages 1828–1833, 1828.

[103] Netty-Community. Netty. https://github.com/netty/netty. Accessed: 2025-03-21.

[104] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

[105] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E Hassan, et al. Multi-granularity detector for vulnerability fixes. *IEEE Transactions on Software Engineering*, 49(8):4035–4057, 2023.

[106] Apache NiFi-Community. Apache nifi. https://github.com/apache/nifi. Accessed: 2025-03-21.

[107] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the ACM conference on Computer and communications security*, pages 736–747, 2012.

[108] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024.

[109] Obieda Ananbeh, Wala Alnozami, Dae-Kyoo Kim, Hua Ming, Weifeng Pan. Java Vulnerability Dataset. https://figshare.com/s/e07242d5b0538be41a4c. Accessed: 2024-06-05.

[110] Ofbiz. ofbiz-framework GitHub Repository. https://github.com/apache/ofbiz-framework. Accessed: 2024-08-1.

[111] Fernando Rodriguez Olivera. Maven repository. https://mvnrepository.com. Accessed: 2024-05-9.

[112] ollama. ollama. https://ollama.com. Accessed: 2025-03-23.

[113] OpenAI. Chatgpt: A language model for dialogue generation. Available online, 2022.

[114] Apache OpenMeetings-Community. Apache openmeetings. https://github.com/apache/openmeetings. Accessed: 2025-03-21.

[115] OpenMRS. OpenMRS Core GitHub Repository. https://github.com/openmrs/openmrs-core. Accessed: 2024-04-9.

[116] OpenOLAT. OpenOLAT GitHub Repository. https://github.com/OpenOLAT/OpenOLAT. Accessed: 2024-08-1.

[117] OpenRefine. Openrefine. https://github.com/OpenRefine/OpenRefine. Accessed: 2025-03-21.

[118] Openremote. openremote GitHub Repository. https://github.com/openremote/openremote. Accessed: 2024-08-1.

[119] Ranindya Paramitha and Yudistira Dwi Wardhana Asnar. Mining software repository for security smell code review. In *2021 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2021.

[120] Xin Peng, Shangwen Wang, Yihao Qin, Bo Lin, Liqian Chen, and Xiaoguang Mao. Keep it simple: Towards accurate vulnerability detection for large code graphs. *arXiv preprint arXiv:2412.10164*, 2024.

[121] Eduard Pinconschi, Rui Abreu, and Pedro Adão. A comparative study of automatic program repair techniques for security vulnerabilities. In *2021 IEEE 32nd international symposium on software reliability engineering (ISSRE)*, pages 196–207, 2021.

[122] Portfolio. portfolio GitHub Repository. https://github.com/portfolio-performance/portfolio. Accessed: 2024-08-1.

[123] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[124] Robert C Seacord. Java deserialization vulnerabilities and mitigations. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 6–7, 2017.

[125] Apache Log4j Security. cve-2021-44228. Available online, 2021.

[126] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Computing Surveys (CSUR)*, 44(3):1–46, 2012.

[127] Zhengxiang Shi and Aldo Lipani. Dept: Decomposed prompt tuning for parameter-efficient fine-tuning. *arXiv preprint arXiv:2309.05173*, 2023.

[128] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming Jiang, and Nachiappan Nagappan. A systematic literature review on automated software vulnerability detection using machine learning. *ACM Computing Surveys*, 57(3):1–36, 2024.

[129] Shopizer. shopizer GitHub Repository. https://github.com/shopizer-ecommerce/shopizer. Accessed: 2024-08-1.

[130] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, page 858–870, 2016.

[131] Smile. smile GitHub Repository. https://github.com/Haifengl/smile. Accessed: 2024-08-1.

[132] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543, 2015.

[133] Snyk. Snyk. https://snyk.io. Accessed: 2024-05-25.

[134] Snyk. Snyk security. https://security.snyk.io. Accessed: 2024-02-5.

[135] Tim Sonnekalb, Thomas S Heinze, and Patrick Mäder. Deep security analysis of program code: A systematic literature review. *Empirical Software Engineering*, 27(1):2, 2022.

[136] Haotian Sun, Yuchen Zhuang, Wei Wei, Chao Zhang, and Bo Dai. Bbox-adapter: Lightweight adapting for black-box large language models. *arXiv preprint arXiv:2402.08219*, 2024.

[137] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in neural information processing systems*, 32, 2019.

[138] Synk. Integrate with snyk - jetbrains plugins. https://docs.snyk.io/integrate-with-snyk/use-snyk-in-your-ide/jetbrains-plugins. Accessed: 2024-03-26.

[139] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[140] OpenTSDB Team. Opentsdb. https://github.com/OpenTSDB/opentsdb. Accessed: 2025-03-21.

[141] Teammates. teammates GitHub Repository. https://github.com/TEAMMATES/teammates. Accessed: 2024-08-1.

[142] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19–36, 2018.

[143] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 481–496, 2022.

[144] Thingsboard. thingsboard GitHub Repository. https://github.com/thingsboard/thingsboard. Accessed: 2024-08-1.

[145] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[146] Paul Trust and Rosane Minghim. A study on text classification in the age of large language models. *Machine Learning & Knowledge Extraction*, 6(4), 2024.

[147] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, pages 151–162, 2018.

[148] R Vinayakumar, Mamoun Alazab, KP Soman, Prabaharan Poornachandran, and Sitalakshmi Venkatraman. Robust intelligent malware detection using deep learning. *IEEE access*, 7:46717–46738, 2019.

[149] V Vippalapalli, AS Harshini, G Kaveri, TS Meghana, and G Saisri. Web security audit and penetration testing: Identifying vulnerabilities and strengthening website security. *International Journal for Research in Applied Science and Engineering Technology*, 11:794–805, 2023.

[150] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

[151] Pingyan Wang, Shaoying Liu, Ai Liu, and Wen Jiang. Detecting security vulnerabilities with vulnerability nets. *Journal of Systems and Software*, 208:111902, 2024.

[152] Qian Wang, Zhengdao Li, Hetong Liang, Xiaowei Pan, Hui Li, Tingting Li, Xiaochen Li, Chenchen Li, and Shikai Guo. Graph confident learning for software vulnerability detection. *Engineering Applications of Artificial Intelligence*, 133:108296, 2024.

[153] Ruoke Wang, Zongjie Li, Chaozheng Wang, Yang Xiao, and Cuiyun Gao. Navrepair: Node-type aware c/c++ code vulnerability repair. *arXiv preprint arXiv:2405.04994*, 2024.

[154] Wenbo Wang, Tien N Nguyen, Shaohua Wang, Yi Li, Jiyuan Zhang, and Aashish Yadavally. Deepvd: Toward class-separation features for neural network vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2249–2261, 2023.

[155] Wire. Wire secure messaging. https://github.com/wireapp/wire. Accessed: 2025-03-21.

[156] Yi Wu, Nan Jiang, H. Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[157] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1282–1294, 2023.

[158] Jie Xu, Karthikeyan Saravanan, Rogier van Dalen, Haaris Mehmood, David Tuckey, and Mete Ozay. Dp-dylora: Fine-tuning transformer-based models on-device under differentially private federated learning using dynamic low-rank adaptation. *arXiv preprint arXiv:2405.06368*, 2024.

[159] Yanjing Yang, Xin Zhou, Runfeng Mao, Jinwei Xu, Lanxin Yang, Yu Zhang, Haifeng Shen, and He Zhang. Dlap: A deep learning augmented large language model prompting framework for software vulnerability detection. *Journal of Systems and Software*, 219:112234, 2025.

[160] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing*, 21(4):2507–2525, 2023.

[161] Renrui Zhang, Jiaming Han, Chris Liu, Peng Gao, Aojun Zhou, Xiangfei Hu, Shilin Yan, Pan Lu, Hongsheng Li, and Yu Qiao. Llama-adapter: Efficient fine-tuning of language models with zero-init attention. *arXiv preprint arXiv:2303.16199*, 2023.

[162] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–702, 2022.

[163] Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2565–2577, 2023.

[164] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 2024.

[165] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[166] Zhenxun Zhuang, Mingrui Liu, Ashok Cutkosky, and Francesco Orabona. Understanding adamw through proximal methods and scale-freeness. *arXiv preprint arXiv:2202.00089*, 2022.

[167] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. $\mu$ vuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2019.

# APPENDIX A

## ANALYSIS OF REMAINING VULNERABILITIES BY PROJECT

Table A.1
Remaining Vulnerabilities by Project

| Project | Count | Test related | JUnit and Dependency Related | version |
|---|---|---|---|---|
| zookeeper | 2 | CWE-400 | com.sun.jersey:jersey-json | 1.1.5.1 |
| | 2 | CWE-798 | Junit | NA |
| | 1 | CWE-552 | com.google.guava:guava | 30 |
| | 1 | CWE-121 | com.sun.jersey:jersey-json | 1.1.5.1 |
| | 1 | CWE-173 | com.google.guava:guava | 30 |
| | 1 | CWE-200 | com.google.guava:guava | 30 |
| | 1 | CWE-378 | com.google.guava:guava | 30 |
| | 1 | CWE-379 | com.google.guava:guava | 30 |
| | 1 | CWE-502 | org.apache.rat:apache-rat-tasks | 0.6 |
| | 1 | CWE-732 | com.google.guava:guava | 30 |
| Weasis | 10 | CWE-611 | Junit | NA |
| | 1 | CWE-798 | Junit | NA |
| ThingsBoard | 12 | CWE-798 | Junit | NA |
| | 2 | CWE-269 | com.datastax.oss:java-driver-core | 4.17.0 |
| | 2 | CWE-284 | com.datastax.oss:java-driver-core | 4.17.0 |
| teammates | 4 | CWE-770 | org.apache.solr:solr-solrj | 8.11.3 |
| | 3 | CWE-835 | com.sendgrid:sendgrid-java | 4.10.2 |
| | 2 | CWE-378 | com.mailjet:mailjet-client | 5.2.5 |
| | 1 | CWE-770 | com.google.firebase:firebase-admin | 9.2.0 |
| Smile | 1 | CWE-798 | Junit | NA |
| Shopizer | 5 | CWE-400 | org.apache.tomcat.embed:tomcat-embed-jasper | 9.0.60 |
| | 3 | CWE-400 | mysql:mysql-connector-java | 8.0.21 |
| | 3 | CWE-798 | Junit | NA |
| | 3 | CWE-94 | com.h2database:h2 | 1.4.200 |
| | 2 | CWE-285 | org.springframework.boot:spring-boot-starter-security | 2.5.12 |
| | 1 | CWE-284 | org.springframework.boot:spring-boot-starter-actuator | 2.5.12 |
| portfolio | 1 | CWE-611 | Junit | NA |
| OpenRemote | 4 | CWE-611 | Junit | NA |
| | 3 | CWE-601 | org.springframework:spring-web | 5.3.30 |
| | 3 | CWE-94 | taglibs:standard | 1.1.2 |
| | 2 | CWE-400 | mysql:mysql-connector-java | 8.0.30 |
| | 2 | CWE-798 | Junit | NA |
| OpenOlat | 26 | CWE-798 | Junit | NA |
| | 2 | CWE-256 | Junit | NA |
| | 1 | CWE-378 | com.microsoft.graph:microsoft-graph | 6.13.0 |
| | 1 | CWE-401 | org.apache.cxf:cxf-rt-transports-http | 4.0.4 |
| OpenMRS | 25 | CWE-798 | Junit | NA |
| | 3 | CWE-400 | mysql:mysql-connector-java | 8.0.30 |
| | 3 | CWE-601 | org.springframework:spring-web | 5.3.30 |
| | 3 | CWE-611 | Junit | NA |
| | 2 | CWE-404 | org.apache.tomcat:tomcat-jasper | 8.5.71 |
| | 1 | CWE-20 | org.apache.velocity:velocity-tools | 2.0.1 |
| HAPI-FHIR | 16 | CWE-798 | Junit | NA |
| | 2 | CWE-611 | Junit | NA |
| Encog | 2 | CWE-611 | Junit | NA |
| Eclipse-Kura | 16 | CWE-798 | Junit | NA |
| | 7 | CWE-400 | log4j:log4j | 1.2.17 |
| | 7 | CWE-502 | log4j:log4j | 1.2.17 |
| | 2 | CWE-89 | log4j:log4j | 1.2.17 |

**Table A.1 – Continued**

| Project | Count | Test related | JUnit and Dependency Related | version |
|---|---|---|---|---|
| | 1 | CWE-502 | io.moquette:moquette-broker | 0.15 |
| | 1 | CWE-89 | io.moquette:moquette-broker | 0.15 |
| | 1 | CWE-256 | Junit | NA |
| | 1 | CWE-444 | io.moquette:moquette-broker | 0.15 |
| CoreNLP | 10 | CWE-798 | Junit | NA |
| Broadleaf | 9 | CWE-798 | Junit | NA |
| | 7 | CWE-256 | Junit | NA |
| | 2 | CWE-502 | Junit | NA |
| bookkeeper | 13 | CWE-798 | Junit | NA |
| | 2 | CWE-770 | io.vertx:vertx-core | 4.3.8 |
| bitcoinj | 5 | CWE-798 | Junit | NA |
| Apache OFBiz | 3 | CWE-611 | Junit | NA |
| | 3 | CWE-798 | Junit | NA |
| | 1 | CWE-256 | Junit | NA |
| Apache Ignite | 17 | CWE-798 | Junit | NA |
| | 2 | CWE-200 | io.opencensus:opencensus-exporter-stats-prometheus | 0.31.1 |
| | 2 | CWE-400 | com.fasterxml.woodstox:woodstox-core | 5.0.3 |
| | 2 | CWE-611 | org.yardstickframework:yardstick | 0.8.3 |
| | 2 | CWE-94 | com.h2database:h2 | 1.4.197 |
| | 2 | CWE-20 | org.apache.maven.surefire:maven-surefire-common | 3.1.2 |
| | 2 | CWE-200 | org.apache.maven.surefire:maven-surefire-common | 3.1.2 |
| | 2 | CWE-275 | com.h2database:h2 | 1.4.197 |
| | 2 | CWE-494 | com.beust:jcommander | 1.32 |
| | 2 | CWE-611 | com.fasterxml.woodstox:woodstox-core | 5.0.3 |
| | 1 | CWE-200 | junit:junit | 4.12 |
| | 1 | CWE-77 | org.apache.maven.surefire:maven-surefire-common | 3.1.2 |
| | 1 | CWE-400 | mysql:mysql-connector-java | 8.0.30 |
| Apache Open Meetings | 6 | CWE-379 | org.asteriskjava:asterisk-java | 3.39.0 |
| | 5 | CWE-401 | org.apache.cxf:cxf-rt-frontend-jaxrs | 4.0.4 |
| | 2 | CWE-601 | org.springframework:spring-web | 6.1.5 |
| | 1 | CWE-77 | org.apache.wicket:wicket-core | 10.0.0 |
| | 1 | CWE-77 | org.apache.wicket:wicket-util | 10.0.0 |
| | 1 | CWE-798 | Junit | NA |
| | 1 | CWE-835 | org.apache.tika:tika-parsers-standard-package | 3.0.0-BETA |
| | 1 | CWE-918 | org.apache.cxf:cxf-rt-rs-service-description | 4.0.4 |
| | 1 | CWE-94 | com.h2database:h2 | 2.2.224 |
| Apache Fineract | 17 | CWE-798 | Junit | NA |
| | 1 | CWE-256 | Junit | NA |

# APPENDIX B

## ANALYSIS OF REMAINING VULNERABILITIES BY DOMAIN

Table B.1
Remaining Vulnerabilities by Domain

| Domain | count | Test related | JUnit and Dependency Related | version |
|---|---|---|---|---|
| AI | 10 | CWE-798 | Junit | NA |
| | 2 | CWE-611 | Junit | NA |
| | 1 | CWE-798 | Junit | NA |
| Distributed Systems | 17 | CWE-798 | Junit | NA |
| | 13 | CWE-798 | Junit | NA |
| | 2 | CWE-200 | io.opencensus:opencensus-exporter-stats-prometheus | 0.31.1 |
| | 2 | CWE-400 | com.fasterxml.woodstox:woodstox-core | 5.0.3 |
| | 2 | CWE-611 | org.yardstickframework:yardstick | 0.8.3 |
| | 2 | CWE-770 | io.vertx:vertx-core | 4.3.8 |
| | 2 | CWE-94 | com.h2database:h2 | 1.4.197 |
| | 2 | CWE-20 | org.apache.maven.surefire:maven-surefire-common | 3.1.2 |
| | 2 | CWE-200 | org.apache.maven.surefire:maven-surefire-common | 3.1.2 |
| | 2 | CWE-275 | com.h2database:h2 | 1.4.197 |
| | 2 | CWE-400 | com.sun.jersey:jersey-json | 1.1.5.1 |
| | 2 | CWE-494 | com.beust:jcommander | 1.32 |
| | 2 | CWE-611 | com.fasterxml.woodstox:woodstox-core | 5.0.3 |
| | 2 | CWE-798 | Junit | NA |
| | 1 | CWE-552 | com.google.guava:guava | 30 |
| | 1 | CWE-77 | org.apache.maven.surefire:maven-surefire-common | 3.1.2 |
| | 1 | CWE-121 | com.sun.jersey:jersey-json | 1.1.5.1 |
| | 1 | CWE-173 | com.google.guava:guava | 30 |
| | 1 | CWE-200 | com.google.guava:guava | 30 |
| | 1 | CWE-378 | com.google.guava:guava | 30 |
| | 1 | CWE-379 | com.google.guava:guava | 30 |
| | 1 | CWE-400 | mysql:mysql-connector-java | 8.0.30 |
| | 1 | CWE-502 | org.apache.rat:apache-rat-tasks | 0.6 |
| | 1 | CWE-732 | com.google.guava:guava | 30 |
| | 1 | CWE-200 | junit:junit | 4.12 |
| E-commerce | 9 | CWE-798 | Junit | NA |
| | 7 | CWE-256 | Junit | NA |
| | 5 | CWE-400 | org.apache.tomcat.embed:tomcat-embed-jasper | 9.0.60 |
| | 3 | CWE-400 | mysql:mysql-connector-java | 8.0.21 |
| | 3 | CWE-611 | Junit | NA |
| | 3 | CWE-798 | Junit | NA |
| | 3 | CWE-798 | Junit | NA |
| | 3 | CWE-94 | com.h2database:h2 | 1.4.200 |
| | 2 | CWE-285 | org.springframework.boot:spring-boot-starter-security | 2.5.12 |
| | 2 | CWE-502 | Junit | NA |
| | 1 | CWE-284 | org.springframework.boot:spring-boot-starter-actuator | 2.5.12 |
| | 1 | CWE-256 | Junit | NA |
| Education | 26 | CWE-798 | Junit | NA |
| | 6 | CWE-835 | org.apache.tika:tika-parsers-standard-package | 3.0.0-BETA |
| | 5 | CWE-798 | Junit | NA |
| | 4 | CWE-770 | org.apache.solr:solr-solrj | 8.11.3 |
| | 3 | CWE-835 | com.sendgrid:sendgrid-java | 4.10.2 |
| | 2 | CWE-378 | com.mailjet:mailjet-client | 5.2.5 |
| | 2 | CWE-256 | Junit | NA |
| | 2 | CWE-77 | org.apache.wicket:wicket-core | 10.0.0 |

**Table B.1 – Continued**

| Domain | count | Test related | JUnit and Dependency Related | version |
|---|---|---|---|---|
| | 1 | CWE-378 | com.microsoft.graph:microsoft-graph | 6.13.0 |
| | 1 | CWE-770 | com.google.firebase:firebase-admin | 9.2.0 |
| | 1 | CWE-379 | org.asteriskjava:asterisk-java | 3.39.0 |
| | 1 | CWE-401 | org.apache.cxf:cxf-rt-frontend-jaxrs | 4.0.4 |
| | 1 | CWE-401 | org.apache.cxf:cxf-rt-transports-http | 4.0.4 |
| | 1 | CWE-601 | org.springframework:spring-web | 6.1.5 |
| | 1 | CWE-77 | org.apache.wicket:wicket-util | 10.0.0 |
| | 1 | CWE-918 | org.apache.cxf:cxf-rt-rs-service-description | 4.0.4 |
| | 1 | CWE-94 | com.h2database:h2 | 2.2.224 |
| Financial | 17 | CWE-798 | Junit | NA |
| | 5 | CWE-798 | Junit | NA |
| | 1 | CWE-256 | Junit | NA |
| | 1 | CWE-611 | Junit | NA |
| IoT | 16 | CWE-798 | Junit | NA |
| | 12 | CWE-798 | Junit | NA |
| | 7 | CWE-400 | log4j:log4j | 1.2.17 |
| | 7 | CWE-502 | log4j:log4j | 1.2.17 |
| | 4 | CWE-611 | Junit | NA |
| | 3 | CWE-601 | org.springframework:spring-web | 5.3.30 |
| | 3 | CWE-94 | taglibs:standard | 1.1.2 |
| | 2 | CWE-269 | com.datastax.oss:java-driver-core | 4.17.0 |
| | 2 | CWE-284 | com.datastax.oss:java-driver-core | 4.17.0 |
| | 2 | CWE-400 | mysql:mysql-connector-java | 8.0.30 |
| | 2 | CWE-798 | Junit | NA |
| | 2 | CWE-89 | log4j:log4j | 1.2.17 |
| | 1 | CWE-502 | io.moquette:moquette-broker | 0.15 |
| | 1 | CWE-89 | io.moquette:moquette-broker | 0.15 |
| | 1 | CWE-256 | Junit | NA |
| | 1 | CWE-444 | io.moquette:moquette-broker | 0.15 |
| Medical | 25 | CWE-798 | Junit | NA |
| | 16 | CWE-798 | Junit | NA |
| | 10 | CWE-611 | Junit | NA |
| | 3 | CWE-400 | mysql:mysql-connector-java | 8.0.30 |
| | 3 | CWE-601 | org.springframework:spring-web | 5.3.30 |
| | 3 | CWE-611 | Junit | NA |
| | 2 | CWE-404 | org.apache.tomcat:tomcat-jasper | 8.5.71 |
| | 2 | CWE-611 | Junit | NA |
| | 1 | CWE-20 | org.apache.velocity:velocity-tools | 2.0.1 |
| | 1 | CWE-798 | Junit | NA |

# APPENDIX C

## ANALYSIS OF REMAINING VULNERABILITIES BY CATEGORIES

Table C.1
Remaining Vulnerabilities by Categories

| Vulnerability Category | CWE ID | Count | Dependency Name | Version |
|---|---|---|---|---|
| IER | CWE-798 | 125 | Junit | NA |
| | CWE-256 | 11 | Junit | NA |
| | CWE-200 | 10 | com.google.guava:guava, junit:junit | 30, 4.12 |
| | CWE-378 | 3 | com.google.guava:guava | 30 |
| | CWE-379 | 3 | com.google.guava:guava, org.asteriskjava: asterisk-java | 30, 3.39.0 |
| | CWE-552 | 2 | com.google.guava:guava | 30 |
| | CWE-285 | 2 | org.springframework.boot: spring-boot-starter-security | 2.5.12 |
| | CWE-269 | 2 | com.datastax.oss: java-driver-core | 4.17.0 |
| | CWE-284 | 5 | org.springframework.boot: spring-boot-starter-actuator | 2.5.12 |
| INH | CWE-770 | 7 | io.vertx:vertx-core, org.apache.solr:solr-solrj | 4.3.8, 8.11.3 |
| ITH | CWE-173 | 2 | com.google.guava:guava | 30 |
| | CWE-835 | 9 | org.apache.tika: tika-parsers-standard-package | 3.0.0-BETA |
| ITV | CWE-275 | 4 | com.h2database:h2 | 1.4.197 |
| | CWE-732 | 2 | com.google.guava:guava | 30 |
| | CWE-494 | 1 | com.beust:jcommander | 1.32 |
| LOBC | CWE-400 | 15 | com.sun.jersey:jersey-json, mysql:mysql-connector-java | 1.1.5.1, 8.0.30 |
| | CWE-404 | 3 | org.apache.tomcat:tomcat-jasper | 8.5.71 |
| | CWE-94 | 9 | com.h2database:h2 | 1.4.197, 2.2.224 |
| LOWV | CWE-611 | 20 | Junit | NA |
| | CWE-444 | 10 | io.moquette:moquette-broker | 0.15 |
| | CWE-611 | 4 | com.fasterxml.woodstox: woodstox-core | 5.0.3 |
| IAC | CWE-121 | 5 | com.sun.jersey:jersey-json | 1.1.5.1 |
| LOWV, ITH | CWE-20 | 9 | org.apache.maven.surefire: maven-surefire-common | 3.1.2 |
| LOIS, IER | CWE-89 | 9 | io.moquette:moquette-broker | 0.85 |
| LOIS, ITV | CWE-502 | 15 | io.moquette:moquette-broker | 0.15 |
| LOIS, LOWV | CWE-601 | 7 | org.springframework:spring-web | 5.3.30, 6.1.5 |
| LOIS, LOWV | CWE-77 | 3 | org.apache.wicket:wicket-core | 10.0.0 |

# APPENDIX D

## IDENTIFICATION AND FIX DISTRIBUTION BY CWE TYPES

Table D.1
Identification distribution by CWE types

| CWE ID | VulnFixAI | ChatGPT 4 | Claude 3.5 Sonnet | Gemini 2.0 Flash | llama3.2 (3B) |
|---|---|---|---|---|---|
| CWE-23 | 499 | 421 | 414 | 413 | 392 |
| CWE-319 | 375 | 353 | 350 | 331 | 208 |
| CWE-470 | 191 | 179 | 174 | 174 | 175 |
| CWE-918 | 187 | 164 | 159 | 139 | 130 |
| CWE-79 | 138 | 128 | 125 | 120 | 116 |
| CWE-611 | 95 | 83 | 82 | 81 | 79 |
| CWE-916 | 78 | 66 | 55 | 55 | 56 |
| CWE-113 | 69 | 51 | 51 | 50 | 55 |
| CWE-295 | 43 | 33 | 31 | 31 | 30 |
| CWE-89 | 41 | 29 | 29 | 28 | 33 |
| CWE-78 | 32 | 26 | 26 | 24 | 25 |
| CWE-614 | 28 | 20 | 20 | 20 | 23 |
| CWE-601 | 26 | 18 | 18 | 18 | 20 |
| CWE-352 | 21 | 15 | 15 | 15 | 16 |
| CWE-502 | 13 | 9 | 9 | 9 | 12 |
| CWE-134 | 12 | 9 | 9 | 9 | 10 |
| CWE-208 | 11 | 9 | 9 | 7 | 7 |
| CWE-732 | 11 | 8 | 8 | 8 | 8 |
| CWE-326 | 10 | 9 | 8 | 8 | 7 |
| CWE-327 | 8 | 7 | 5 | 5 | 5 |
| CWE-501 | 4 | 3 | 3 | 3 | 3 |
| CWE-90 | 3 | 2 | 2 | 2 | 2 |
| CWE-547 | 2 | 2 | 2 | 2 | 2 |

Table D.2
Fix distribution by CWE types

| CWE ID | VulnFixAI | ChatGPT 4 | Claude 3.5 Sonnet | Gemini 2.0 Flash | llama3.2 (1B) |
|---|---|---|---|---|---|
| CWE-23 | 461 | 356 | 343 | 337 | 289 |
| CWE-319 | 357 | 311 | 287 | 266 | 234 |
| CWE-918 | 179 | 140 | 134 | 112 | 99 |
| CWE-470 | 171 | 146 | 137 | 135 | 118 |
| CWE-79 | 133 | 109 | 105 | 101 | 79 |
| CWE-611 | 86 | 71 | 68 | 68 | 62 |
| CWE-113 | 66 | 43 | 43 | 42 | 38 |
| CWE-916 | 52 | 50 | 48 | 48 | 43 |
| CWE-295 | 39 | 28 | 27 | 26 | 24 |
| CWE-78 | 31 | 21 | 20 | 20 | 18 |
| CWE-601 | 26 | 16 | 16 | 15 | 15 |
| CWE-614 | 22 | 17 | 17 | 17 | 16 |
| CWE-352 | 19 | 13 | 12 | 12 | 11 |
| CWE-89 | 16 | 26 | 26 | 25 | 23 |
| CWE-134 | 12 | 8 | 8 | 8 | 8 |
| CWE-208 | 11 | 7 | 7 | 7 | 7 |
| CWE-326 | 9 | 7 | 7 | 7 | 7 |
| CWE-502 | 8 | 9 | 9 | 9 | 9 |
| CWE-327 | 5 | 5 | 5 | 5 | 5 |
| CWE-501 | 4 | 3 | 2 | 2 | 2 |
| CWE-90 | 3 | 2 | 2 | 2 | 2 |
| CWE-547 | 1 | 2 | 2 | 2 | 2 |
| CWE-732 | 0 | 7 | 7 | 7 | 6 |

# APPENDIX E

# IDENTIFICATION AND FIX DISTRIBUTION ACROSS PROJECTS AND CWE TYPES

Table E.1
Identification Results of Top 5 CWE Types Across Projects

| Project Name | CWE ID | VulnFixAl | ChatGPT 4 | Claude 3.5 Sonnet | Gemini 2.0 Flash | llama3.2 (3B) |
|---|---|---|---|---|---|---|
| Eclipse GlassFish | CWE-23 | 187 | 181 | 178 | 180 | 166 |
| Apache NiFi | CWE-23 | 56 | 43 | 42 | 42 | 41 |
| Apache Hadoop | CWE-23 | 51 | 39 | 38 | 38 | 37 |
| Apache CXF | CWE-23 | 50 | 38 | 38 | 37 | 36 |
| Apache Flink | CWE-23 | 28 | 21 | 21 | 21 | 20 |
| Eclipse GlassFish | CWE-319 | 298 | 292 | 290 | 276 | 157 |
| Apache NiFi | CWE-319 | 35 | 27 | 31 | 26 | 24 |
| James | CWE-319 | 9 | 7 | 7 | 7 | 6 |
| undertow | CWE-319 | 9 | 7 | 7 | 7 | 6 |
| Apache Flink | CWE-319 | 6 | 5 | 4 | 4 | 4 |
| Apache Hadoop | CWE-470 | 152 | 149 | 144 | 144 | 145 |
| Apache Hadoop | CWE-470 | 15 | 11 | 11 | 11 | 11 |
| Eclipse GlassFish | CWE-470 | 8 | 6 | 6 | 6 | 6 |
| HyperSQL | CWE-470 | 5 | 4 | 4 | 4 | 4 |
| Eclipse GlassFish | CWE-470 | 4 | 3 | 3 | 3 | 3 |
| Eclipse GlassFish | CWE-918 | 121 | 112 | 110 | 90 | 84 |
| Apache CXF | CWE-918 | 23 | 17 | 17 | 17 | 16 |
| Apache Hadoop | CWE-918 | 16 | 12 | 12 | 12 | 11 |
| Keycloak | CWE-918 | 12 | 9 | 9 | 9 | 8 |
| Apache CXF | CWE-918 | 4 | 3 | 3 | 3 | 3 |
| Eclipse GlassFish | CWE-79 | 108 | 106 | 104 | 99 | 92 |
| openmeetings | CWE-79 | 12 | 9 | 9 | 9 | 9 |
| Keycloak | CWE-79 | 7 | 5 | 5 | 5 | 5 |
| Apache Dubbo | CWE-79 | 2 | 1 | 1 | 1 | 2 |
| FitNesse | CWE-79 | 2 | 2 | 1 | 1 | 1 |

Table E.2
Identification Results of Top 5 CWE Types Across Projects

| Project Name | CWE ID | VulnFixAI | llama3.2 (1B) | ChatGPT 4 | Claude 3.5 Sonnet | Gemini 2.0 Flash Fix |
|---|---|---|---|---|---|---|
| Eclipse GlassFish | CWE-23 | 187 | 121 | 157 | 153 | 150 |
| Apache NiFi | CWE-23 | 56 | 30 | 36 | 34 | 33 |
| Apache Hadoop | CWE-23 | 51 | 27 | 32 | 31 | 30 |
| Apache CXF | CWE-23 | 50 | 26 | 32 | 31 | 30 |
| Apache Flink | CWE-23 | 28 | 15 | 18 | 17 | 17 |
| Eclipse GlassFish | CWE-319 | 298 | 193 | 262 | 237 | 221 |
| Apache NiFi | CWE-319 | 35 | 18 | 22 | 26 | 21 |
| James | CWE-319 | 9 | 5 | 6 | 5 | 5 |
| Apache Flink | CWE-319 | 6 | 3 | 4 | 4 | 4 |
| FitNesse | CWE-319 | 4 | 2 | 3 | 2 | 2 |
| Eclipse GlassFish | CWE-918 | 121 | 63 | 96 | 93 | 72 |
| Apache CXF | CWE-918 | 23 | 12 | 15 | 14 | 14 |
| Apache Hadoop | CWE-918 | 16 | 8 | 10 | 10 | 9 |
| Keycloak | CWE-918 | 12 | 6 | 8 | 7 | 7 |
| Hugegraph-toolchain | CWE-918 | 2 | 1 | 1 | 1 | 1 |
| Apache Hadoop | CWE-470 | 152 | 95 | 121 | 112 | 110 |
| Eclipse GlassFish | CWE-470 | 8 | 4 | 5 | 5 | 5 |
| HyperSQL | CWE-470 | 5 | 3 | 3 | 3 | 3 |
| Keycloak | CWE-470 | 3 | 2 | 2 | 2 | 2 |
| Apache CXF | CWE-470 | 1 | 1 | 1 | 1 | 1 |
| Eclipse GlassFish | CWE-79 | 108 | 64 | 92 | 89 | 83 |
| openmeetings | CWE-79 | 12 | 6 | 7 | 7 | 7 |
| Keycloak | CWE-79 | 7 | 4 | 4 | 4 | 4 |
| FitNesse | CWE-79 | 2 | 0 | 1 | 0 | 1 |
| OpenRefine | CWE-79 | 2 | 1 | 1 | 1 | 1 |