FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Evaluating Static Analysis Tools in Identifying Software Vulnerabilities at the Root-Cause Level: A Machine-Learning-Based Triage Approach

Eduard Costel Pinconschi



Programa Doutoral em Engenharia Informática

Supervisor: Prof. Rui Abreu (FEUP)

Co-Supervisor: Prof. Pedro Adão (IST)

Co-Supervisor: Prof. Corina Păsăreanu (CMU)

Co-Supervisor: Prof. Hakan Erdogmus (CMU)

February 26, 2026

© Eduard Pinconschi, 2026

Evaluating Static Analysis Tools in Identifying Software Vulnerabilities at the Root-Cause Level: A Machine-Learning-Based Triage Approach

Eduard Costel Pinconschi

Programa Doutoral em Engenharia Informática

Approved in oral examination by the committee on October 2, 2026:

President: Prof. Pedro Nuno Ferreira da Rosa da Cruz Diniz
External Member: Dr. Irena Bojanova
External Member: Prof. Nuno Miguel Carvalho Santos
Internal Member: Prof. Ana Cristina Ramada Paiva
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu

February 26, 2026

Resumo

O conhecimento sobre vulnerabilidades ainda é comunicado principalmente por meio do que é mais fácil de observar e triar — falhas (*crashes*), vetores de ataque, e impactos — enquanto a análise de segurança e a correção subsequente exigem uma compreensão ao nível do defeito, fundamentada no código. Esta tese aborda a incompatibilidade entre a comunicação de vulnerabilidades baseada em resultados observáveis e os bugs subjacentes no código que constituem a *ground truth* para detecção, avaliação e reparo. Esta tese investiga três questões: (RQ1) quais perfis caracterizam defeitos de segurança no software quando descritos em termos de mecanismos de causa raiz e contexto de software; (RQ2) quais atributos um *benchmark* deve preservar para que a avaliação distinga causas de manifestações, em vez de recompensar proximidade incidental; e (RQ3) em que medida a triagem baseada em aprendizagem de máquina pode identificar alertas acionáveis de *static application security testing* (SAST) — ferramentas que analisam o código-fonte para sinalizar possíveis falhas de segurança — quando a *ground truth* é definida no nível do defeito em vez do erro.

A tese modela cada vulnerabilidade como uma cadeia *defeito* → *erro* → *falha* e fundamenta a análise no código *open source* como evidência. Os defeitos são reconstruídos a partir de *diffs* de *patches* e de artefatos relacionados, organizados num esquema de caracterização que separa a causa raiz, a manifestação e o contexto de software. Essas reconstruções são operacionalizadas em instâncias executáveis de *benchmark* ao vincular divulgações de *fuzzing* a CVEs, reconstruir ambientes de compilação, reproduzir entradas no programa para validar *crashes* e preservar duas localizações: *bug-sites* localizadas pelo *patch* e *error-sites* localizadas pela falha. Com base nesse *benchmark*, a tese avalia quatro ferramentas de *SAST* nas duas localizações ao nível da linha (*bug site* versus *error site*), avalia a semântica reportada pelas ferramentas por meio de abstrações ao nível da operação e introduz uma camada de triagem baseada em explicações que complementa os alertas e treina um classificador leve para filtrar alertas não acionáveis.

A análise de 55 vulnerabilidades reconstruídas mostra que os defeitos formam perfis estáveis e interpretáveis quando condicionados pela causa (*Missing* versus *Incorrect*) e pela operação no contexto do software, e que resultados semelhantes, de alta severidade, podem surgir por meio de regimes de erro e de perfis de bug distintos. Dos 69 casos derivados de *fuzzing*, 30 são convertidos em instâncias de *benchmark* que reproduzem *crashes*; estas mostram que as localizações do bug e do erro frequentemente divergem e que as operações do bug são diversas, mas convergem para um conjunto restrito de operações de erro. Ao avaliar as ferramentas de *SAST* nas duas localizações ao nível da linha (*bug site* e *error site*), observa-se um sobre-relato extremo, e a deteção ocorre com maior frequência nos *error sites* do que nos *bug sites*. Mesmo quando os alertas coincidem com uma das duas localizações, a sua avaliação, por meio da operação, indica deteção incidental. A triagem baseada em aprendizagem de máquina reduz a carga de falsos positivos, preservando, de forma mais consistente, alertas que correspondem a defeitos, em vez de manifestações.

De modo geral, a tese demonstra que uma avaliação fiável de *SAST* exige a modelação explícita das vulnerabilidades ao nível da linha, distinguindo entre defeitos e erros, e avaliando os alertas

com base na semântica das operações. A tese contribui com uma metodologia de caracterização de defeitos fundamentada no *patch*, um *benchmark* com instâncias reprodutíveis anotadas em ambas as localizações e uma abordagem de avaliação que considera a separação entre defeitos e suas manifestações, estabelecendo uma base para avanços mais fundamentados na direção da deteção acionável e do reparo de vulnerabilidades.

Abstract

Vulnerability knowledge is still communicated primarily through what is easiest to observe and triage—crashes, attack vectors, and impacts—while security analysis and downstream repair require root cause understanding grounded in code. This thesis addresses the mismatch between vulnerability reporting based on observable outcomes and the underlying bugs in code that constitute the ground truth for detection, evaluation, and repair. It investigates three questions: (RQ1) what profiles characterize software security bugs when described in terms of root-cause behaviour and software context; (RQ2) which attributes a benchmark must preserve so that evaluation distinguishes causes from manifestations rather than rewarding incidental proximity; and (RQ3) to what extent machine learning–based triage can identify actionable warnings from static application security testing (SAST) — tools that analyze source code to flag potential security flaws — when ground truth is defined at the level of the bug instead of error.

The thesis models each vulnerability as a *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chain and grounds the analysis in open-source evidence. Bugs are reconstructed from patch diffs and related artifacts, then organized into a profiling schema that separates root cause, manifestation, and software context. These reconstructions are operationalized into executable benchmark instances by linking fuzzing disclosures to CVEs, reconstructing build environments, replaying program inputs to validate crashes, and preserving two locations: patch-localized *bug sites* and crash-localized *error sites*. Using this benchmark, the thesis evaluates four SAST tools under the two locations at the line-level (bug-site vs. error-site), assesses tool-reported semantics through operation-level abstractions, and introduces an explanation-based triage layer that augments warnings and trains a lightweight classifier to filter non-actionable warnings.

Analysis of 55 reconstructed vulnerabilities shows that bugs form stable, interpretable profiles when conditioned by cause (*Missing* vs. *Incorrect*) and by operation within the software context, and that similar high-severity outcomes can arise through distinct error regimes and bug profiles. From 69 linked fuzzing cases, 30 are converted into benchmark instances that reproduce crashes; these show that bug and error locations frequently diverge and that the bug operations are diverse but converge into a narrow set of error operations. Evaluating the SAST tools in both locations at the line-level (bug site and error site), these exhibit extreme over-reporting, and their detection occurs more often at error sites than at bug sites. Even when warnings match one of the two locations, their assessment (through the operation) indicates incidental detection. Machine learning–based triage reduces the burden of false positives while preserving warnings that correspond more reliably to bugs than to their manifestations.

Overall, the thesis shows that reliable SAST evaluation requires explicit modeling of vulnerabilities at the line level, distinguishing between bugs and errors, and assessing warnings using operation semantics. It contributes a bug-profiling methodology grounded in patch artifacts, a benchmark with reproducible instances annotated at both locations, and an evaluation approach that accounts for the separation between bugs and their manifestations, providing a foundation for more principled progress toward actionable detection and repair of vulnerabilities.

Acknowledgements

Throughout this PhD journey, many people have contributed in different ways, and I am deeply grateful to all who were part of it, especially those who supported me along the way.

From the very beginning—when I filled out the forms and first applied for a PhD—my supervision team was consistently present and supportive. Whether through prompt revisions of administrative documents, long and thoughtful discussions, or guidance through rigorous submissions, your support has been invaluable. I have also greatly benefited from the freedom to explore ideas and pursue opportunities, which has been fundamental to my growth as a researcher and as a person. For all of this, I am sincerely thankful to the four of you; each of you has had a marked impact on this journey. I am aware that I may not always have been the easiest PhD candidate to supervise, and I appreciate your patience, goodwill, and trust throughout these years.

I would also like to thank Professor Onyeka Ezenwoye from Augusta University, who, although not formally part of the supervision team, has been an integral presence since the early years of this PhD. His availability, insight, and willingness to help have been deeply appreciated. My sincere thanks extend to all collaborators with whom I have worked, directly or indirectly, and who contributed to the development of the works—both published and unpublished—that shaped this thesis. I am equally grateful to my colleagues, both in Portugal and in the United States, who made this journey intellectually stimulating and personally enriching. Within the university environment, I thank the staff of the Department of Informatics Engineering for their consistent support throughout the many administrative processes. I also acknowledge the faculty members who, in different ways, have influenced me throughout this journey.

Beyond the university setting, the support and positivity of my friends were a quiet but essential foundation, providing steadiness when it was most needed. Over these years, I have lived in different places and met many remarkable people who left a lasting mark on me—you know who you are. I am especially thankful to those who made difficult moments lighter by bringing kindness, warmth, and, at times, a much-needed smile. Rui Nogueira, you are a particular example of that light, always present with generosity and guidance, like a father figure, despite no family ties. To my family, I am profoundly grateful for your unconditional support, for keeping me grounded, and for providing the stability that allowed me to pursue this path with confidence.

Finally, I gratefully acknowledge the CMU Portugal Program, whose support enabled this PhD and made possible one of the most transformative years of my life: a one-year research period at Carnegie Mellon University under the CMU Portugal Affiliated Ph.D. Programs Initiative. This experience made the journey exponentially more exciting, meaningful, and fulfilling. The number of stories, experiences, and lasting friendships formed during that year would fill the pages of another thesis. The CMU Portugal scholarship was funded by Fundação para a Ciência e a Tecnologia (FCT) through the research grant PRT/BD/152197/2021.

Eduard Pinconschi

"For with much wisdom comes much sorrow; the more knowledge, the more grief."

Ecclesiastes 1:18

Contents

1 Introduction **1**
- 1.1 Problem Context and Motivation . . . 2
- 1.2 Technical and Conceptual Background . . . 4
- 1.3 Research Trajectory and Problem Refinement . . . 6
- 1.4 Thesis Objectives and Research Questions . . . 7
- 1.5 Research Approach and Methodological Strategy . . . 9
- 1.6 Summary of Contributions . . . 10
- 1.7 Structure of the Thesis . . . 12
- 1.8 Relationship to Prior Publications . . . 13

2 Profiling Security Bugs in OSS **15**
- 2.1 Introduction . . . 15
  - 2.1.1 Motivation and gap . . . 15
  - 2.1.2 Bug profiling approach and scope . . . 16
  - 2.1.3 Research questions and contributions . . . 17
  - 2.1.4 Publications context and roadmap . . . 18
- 2.2 Background . . . 18
  - 2.2.1 Software Vulnerabilities and the Bug–Error–Failure Chain . . . 18
  - 2.2.2 Vulnerabilities Across the OSS Lifecycle and Ecosystems . . . 19
  - 2.2.3 Vulnerability Classification Frameworks and Bug Representation . . . 20
  - 2.2.4 Vulnerability Databases and Repositories . . . 22
- 2.3 Related Work . . . 23
  - 2.3.1 Vulnerability Taxonomies in Empirical Analysis . . . 23
  - 2.3.2 Vulnerability Causality and Propagation Chains . . . 25
  - 2.3.3 Automated Analysis and Enrichment of Vulnerability Records . . . 26
  - 2.3.4 Linking Vulnerability Records to Open-Source Repositories . . . 27
  - 2.3.5 Security Patch Identification and Analysis for Bug Localization . . . 28
  - 2.3.6 Summary and Research Gap . . . 30
- 2.4 Methodology . . . 30
  - 2.4.1 Curating and Validating CVE Metadata for Eligibility . . . 31
  - 2.4.2 Filtering CVEs by Exploitability for Code-Level Bugs . . . 32
  - 2.4.3 Delimiting Failure Boundary and OSS Traceability . . . 32
  - 2.4.4 Reconstructing Bug–Error–Failure Chains from Artifacts . . . 34
  - 2.4.5 Vulnerability Representation and Labeling Schema . . . 35
- 2.5 Results . . . 38
  - 2.5.1 Dataset Overview . . . 38
  - 2.5.2 Profiles of Software Security Bugs . . . 39
  - 2.5.3 Manifestation Chains of Software Security Bugs . . . 42

2.6 Discussion . . . 44
2.6.1 Interpretation of results . . . 44
2.6.2 Implications for reporting, research, and practice . . . 46
2.6.3 Threats to validity and limitations . . . 47
2.6.4 Future work . . . 48
2.7 Summary . . . 49

3 Crafting a Vulnerability Benchmark **51**
3.1 Introduction . . . 51
3.1.1 Motivation and gap . . . 51
3.1.2 Benchmark construction approach and scope . . . 52
3.1.3 Research questions and contributions . . . 52
3.1.4 Publications context and roadmap . . . 53
3.2 Background . . . 54
3.2.1 Conceptual Basis: The Defect–Error–Failure Distinction . . . 54
3.2.2 Reproducible Vulnerability Instances . . . 55
3.2.3 Dynamic Analysis and Fuzzing . . . 56
3.2.4 Fault Localization . . . 57
3.3 Related Work . . . 57
3.3.1 Reproducibility, Exploitability, and Vulnerability Benchmarks . . . 58
3.3.2 Fuzzing for Vulnerability Reproduction . . . 59
3.3.3 Crash-Based Statistical Root-Cause Localization via Fuzzing . . . 60
3.3.4 Summary and Research Gap . . . 62
3.4 Methodology . . . 62
3.4.1 Extracting vulnerability reproduction metadata for OSS-Fuzz from OSV . . . 63
3.4.2 Linking OSS-Fuzz Vulnerabilities with CVEs in the NVD . . . 64
3.4.3 Reproducing and Verifying OSS-Fuzz CVEs Automatically . . . 65
3.4.4 Localizing Vulnerability Root Causes from Crash and Patch Artifacts . . . 66
3.5 Results . . . 67
3.5.1 Benchmark overview . . . 67
3.5.2 Reproduction outcomes . . . 68
3.5.3 Function-level and line-level separation between bug and error locations . . . 69
3.5.4 Assessment of the operation at the bug and error locations . . . 70
3.5.5 Cross-reference analysis of separation and assessment . . . 72
3.6 Discussion . . . 72
3.6.1 Interpretation of results . . . 73
3.6.2 Implications for research, tooling, and practice . . . 74
3.6.3 Limitations and threats to validity . . . 75
3.6.4 Future Work . . . 76
3.7 Summary . . . 77

4 Evaluating ML-Based SAST **79**
4.1 Introduction . . . 79
4.1.1 Motivation and gap . . . 79
4.1.2 Evaluation, triage approach, and scope . . . 80
4.1.3 Research question and contributions . . . 81
4.1.4 Publications context and roadmap . . . 82
4.2 Background . . . 82
4.2.1 Static Analysis: Definitions and Foundations . . . 83

4.2.2 Static Analysis for Vulnerability Detection . . . 83
4.2.3 Machine-Learning in Static Analysis and Vulnerability Detection . . . 86
4.2.4 Pretrained Transformers and Large Language Models . . . 87
4.3 Related Work . . . 88
4.3.1 Empirical Studies of SAST Tools for C/C++ Vulnerability Detection . . . 88
4.3.2 Identification of Actionable Warnings through Data-Oriented Classification 91
4.3.3 Generative Reasoning–Based Warning Triage . . . 93
4.3.4 Summary and Research Gap . . . 95
4.4 Methodology . . . 96
4.4.1 Selection, Configuration, and Execution of SAST Tools . . . 96
4.4.2 Constructing a Root Cause Aligned Training Dataset for ML-Based Triage 97
4.4.3 Reasoning Layer: Explanation-Driven Warning Triage . . . 98
4.4.4 Evaluation Design: Detection and Assessment of SAST Findings . . . 99
4.5 Results . . . 101
4.5.1 Experimental setup and datasets . . . 101
4.5.2 Detection of SAST tools under bug and error locations . . . 102
4.5.3 Assessment of SAST tool warnings at detected locations . . . 103
4.5.4 Triage of SAST tool warnings at detected locations . . . 105
4.6 Discussion . . . 106
4.6.1 Interpretation of results . . . 106
4.6.2 Implications for research, tooling, and practice . . . 107
4.6.3 Limitations and threats to validity . . . 108
4.6.4 Future Work . . . 109
4.7 Summary . . . 110

5 Conclusions and Outlook 112
5.1 Revisiting Research Questions . . . 113
5.2 Synthesis of Key Findings . . . 115
5.3 Implications . . . 117
5.4 Limitations and Threats to Validity . . . 119
5.5 Future Directions . . . 120
5.6 Concluding Remarks . . . 121

References 124

A Implementation Details (Chapter 2) 145
A.1 Dataset Construction: Filtering and Curation . . . 145
A.1.1 NVD Status and Tag Filters . . . 145
A.1.2 CPE Dictionary Processing and Consistency Checks . . . 146
A.1.3 CWE Eligibility Rules . . . 147
A.1.4 CVSSv3 Inconsistency Measure . . . 148
A.1.5 CVSSv3 Low-Contingency and Failure-Boundary Filters . . . 149
A.1.6 OSS Repository Resolution and Application-Level Traceability . . . 150
A.1.7 Quality Screening of Descriptions and Extraction of Vulnerability Entities 151
A.1.8 Patch Resolution and Validation . . . 153
A.2 Profiling Labels: Taxonomies and Mappings . . . 154
A.2.1 Root cause taxonomy (cause × operation) and patch evidence . . . 154
A.2.2 Error taxonomy and CWE mapping . . . 157
A.2.3 Failure taxonomy and impact mapping . . . 160

A.2.4 Software Type Taxonomy for Labeling Vulnerable Applications 162
A.2.5 Functional Area Taxonomy for Labeling Vulnerable Software Components 164

B Implementation Details (Chapter 3) 167
B.1 Metadata Validation for OSS-Fuzz OSV Records 167
B.1.1 OSS-Fuzz Issue-Tracker Resolution and Traceability 167
B.1.2 Crash-Type to CWE Mapping (Manifestation-Level) 168
B.1.3 Range Status and Commit Availability 168
B.2 Operational Details of OSS-Fuzz–to–CVE Linkage 169
B.2.1 Product Alignment and Repository–Product Normalization 170
B.2.2 Weakness Alignment via CWE Intersection 170
B.2.3 Reference-Level Provenance Matching 171
B.2.4 Semantic Disambiguation and Priority-Based Match Selection 172
B.3 Automated Reproduction and Fix Verification Procedure 172
B.3.1 Tooling Architecture and Execution Model 173
B.3.2 Context Reconstruction and Snapshot Resolution 173
B.3.3 Containerized Build and Instrumentation Reproduction 174
B.3.4 Crash Signature Parsing and Matching 175
B.4 Patch-Driven Root-Cause Localization and Crash Separation 176
B.4.1 Crash-Context Anchor Extraction and Snapshot-to-Commit Patch Retrieval 176
B.4.2 Diff-Based Root-Cause Selection via Line Proximity 177
B.4.3 Call-Graph Separation Between Bug Location and Error Location 178

C Implementation Details (Chapter 4) 180
C.1 Tool Setup and Scan Pipeline 180
C.1.1 Benchmark / Evaluation Dataset Normalization 180
C.1.2 Execution Environment and Reproducibility Controls 181
C.1.3 Tool Invocation and Input/Output Handling 181
C.2 Training Dataset Construction Pipeline 183
C.2.1 CVE Selection and Patch Constraints 183
C.2.2 Diff Processing and Bug-Fix Extraction 184
C.2.3 SAST-Based Training Data Labeling 185
C.3 Reasoning Layer Implementation and Explanation Generation 185
C.3.1 Context Assembly and Model Invocation 186
C.3.2 Prompt Template and Instruction Variants 187
C.3.3 Response Processing and SARIF Enrichment 190
C.4 Supervised Triage Model and SARIF-Level Filtering 190
C.4.1 Training Dataset Preparation 191
C.4.2 Model Architecture, Configuration, and Training 191
C.4.3 Classifier-Based SARIF Filtering 192
C.4.4 ROC Analysis and Decision Threshold Characterization 193
C.5 Evaluation Protocol and Matching Details 193
C.5.1 Exact-Line Ground-Truth Matching and Warning Deduplication 194
C.5.2 Mapping Tool Rules to Bugs Framework Operations 195

List of Figures

2.1 Manifestation chain of defect, error, and failure (adapted from [10]). 19
2.2 Example of a fault (left) and a security fault (right). 20
2.3 Overview of the methodology: (1) CVE curation, (2) exploitability filtering, (3) failure delimitation and OSS traceability, (4) bug-error-failure chain reconstruction, and (5) labeling the resulting cases by root cause, software context, error, and failure. 31
2.4 Sankey flow summarizing triage of CVEs by *Status*, *CPE*, *CWE*, and *CVSSv3*. . . 32
2.5 Sankey flow summarizing triage of CVEs by *exploitability* CVSSv3 metrics. . . . 33
2.6 Sankey flow summarizing triage of CVEs by *severity* and *OSS traceability*. . . . 33
2.7 CVE description with extracted entities used for reconstruction (CVE-2019-6991). 34
2.8 Patch excerpt for CVE-2019-6991; context lines collapsed. 35
2.9 Sankey flow summarizing triage of CVEs by *description* and *patch* extraction. . . 35
2.10 Manifestation labeling for CVE-2019-6991 based on the CVE description. 36
2.11 Root-cause labeling for CVE-2019-6991 based on the patch analysis. 37
2.12 Software-context labeling for CVE-2019-6991 based on the repository artifacts. . 37
2.13 Distribution of bug operation across software types by cause. Each heatmap reports the concentration of operations by software type, contrasting Missing and Incorrect bugs. 40
2.14 Distribution of bug operation across functional areas by cause. Each heatmap reports the concentration of operations by functional area, contrasting Missing and Incorrect bugs. 41
2.15 Distributions of bug operations across progr. languages by cause. Each heatmap reports the concentration of operations by language, contrasting Missing and Incorrect bugs. 41
2.16 Sankey-style overview of the manifestation chains linking bug cause and operation to propagated errors types and failure outcomes. 43

3.1 Sankey diagram summarizing the filtering of OSS-Fuzz OSV entries by Entry Status, Issue Tracking, CWE Mapping, and Range Status. 63
3.2 Sankey diagram illustrating the OSS-Fuzz to CVE linkage workflow, filtering by product overlap, CWE overlap, and reference evidence leading to final CVE correspondences. 65
3.3 Bug and error separation by call-graph distance (x-axis). Each dot represents one reproduced vulnerability; the y-axis encodes counts. 69
3.4 Within-function bug and error line separation (x-axis on a symmetric log scale to show both small and large separations); each dot represents one reproduced vulnerability; the y-axis encodes counts. 70
3.5 Sankey flow for error assessment: weakness type to error CWE to error operation. 71

3.6 Sankey flow of bug assessment: cause to operation to CWE. 71
3.7 Heatmap cross-referencing bug–error separation (rows) with assessment (columns). 72

4.1 Reported CWEs and respective operations for warnings detecting bug locations. . 104
4.2 Reported CWEs and respective operations for warnings detecting error locations. 104

A.1 CWE Counts Before and After Filtering by Software Assurance Trends Categories. 148
A.2 Hierarchical Regular Expression Extraction Strategy 152

C.1 ROC curves for bug-site and error-site triage. The classifier exhibits strong separability at defect locations (AUC = 0.855) but near-chance performance at error locations (AUC = 0.551). 193

List of Tables

1.1 Contributions organized by their role in the evaluation framework. 11

2.1 Summary Table of Reviewed Taxonomies. 24
2.2 Frequent tuples of the underlying bug profiles. 42
2.3 Baseline disttributions of errors and failures by cause. 43
2.4 Dominant error–failure manifestations enabled by canonical security bug profiles. 44

3.1 Reproduction outcomes and representativeness of reproduced cases. 68

4.1 Static Analyzers used in Empirical Studies [8, 50, 140] 84
4.2 Summary of Empirical Studies that Evaluate SAST Tools on Real C/C++ CVEs . 91
4.3 Warning counts within benchmark-scoped files before and after deduplication. . . 101
4.4 Detection at bug- and error-sites using exact-line matching. 103
4.5 Bug-site triage results on matched files overall and per tool. Precision/Recall/F1 are reported as percentages. 105
4.6 Error-site triage results on matched files overall and per tool. Precision/Recall/F1 are reported as percentages. 105

A.1 CVSSv3 constraints used to select vulnerabilities. 150
A.2 Mapping from representative CWEs to semantic error categories 159
A.3 Software type taxonomy for labeling vulnerable applications in the profiling dataset. 163
A.4 Functional-area taxonomy for labeling vulnerable components. 165

B.1 Additional crash-type to CWE mappings introduced beyond Keller et al. [115]. . 169

C.1 Example training instances derived from the same CVE and SAST rule. The flagged code is syntactically identical, but contextual differences between vulnerable and fixed versions determine the assigned label. 185
C.2 Rule-to-mechanism mappings for analyzer rules that produced location-matched warnings. CWE rationales reflect tool-level weakness interpretation; BF rationales justify abstraction to operation-level semantics. 195

List of Acronyms

AI	Artificial Intelligence
APR	Automatic Program Repair
API	Application Programming Interface
BF	Bugs Framework
CIA	Confidentiality, Integrity, and Availability
CPE	Common Platform Enumeration
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
FL	Fault Localization
ML	Machine Learning
LLM	Large Language Models
NVD	National Vulnerability Database
OS	Operating System
OSS	Open Source Software
OSV	Open Source Vulnerability
POC	Proof-of-Concept
PTM	Pre-Trained Transformer Models
RCA	Root-Cause Analysis
SAST	Static Application Security Testing
SARIF	Static Analysis Results Interchange Format
SBFL	Spectrum-Based Fault Localization

Chapter 1

Introduction

Software vulnerabilities are commonly communicated through what is easiest to observe and triage: failure modes, impacts, and triggering conditions [162, 161]. Preventive techniques and empirical evaluation, however, depend on a different unit of understanding: the bug in code, how it activates into a runtime error, and how that error propagates into an observable failure [10, 16]. This mismatch between reporting based on observable outcomes and reasoning grounded in code creates a structural gap for vulnerability research and evaluation: the information most widely available in public records is often insufficient to define ground truth at the level of the bug [88], yet that is precisely what detection, localization, and downstream repair require [15, 94, 107].

This thesis addresses that gap by establishing empirical and methodological foundations for evaluating vulnerability analysis techniques that distinguish bugs from their manifestations. It focuses on fuzzing-derived security bugs in C/C++ open-source code [202, 154], where reproducible crashes (the failure mode produced by fuzzing [149]) and patch history can be jointly leveraged [80, 81]. Within this scope, vulnerabilities are reconstructed as *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chains [20, 107], with bugs identified from patch artifacts and errors and failures retained as observable manifestations. This yields a representation in which vulnerability location is dual: a *bug line* localized by the fix and an *error line* localized by the crash [15, 107]. This distinction enables evaluation to separate detection of the bug from detection of its manifestation [140, 94].

The thesis is structured around three dependent research questions.

RQ1 asks: *What root causes and contextual factors characterize real-world security bugs, and do stable, interpretable profiles emerge across cases?* It is addressed by reconstructing bug causes and operations in software context and by analyzing how these propagate into errors and failures.

RQ2 asks: *Which vulnerability attributes must a benchmark record for reliable evaluation?* It is addressed by identifying the information a benchmark must retain to preserve the distinction between bug and error locations and to support semantic assessment of tool warnings.

RQ3 asks: *To what extent can SAST outputs, augmented with ML-based triage, identify actionable findings?* It is addressed by evaluating static analysis tools under bug and error locations, assessing whether reported semantics align with bug operations, and examining whether ML-based triage improves warning actionability through explanation-based normalization.

1.1 Problem Context and Motivation

Vulnerability analysis relies on reference information that captures how bugs arise and propagate in real software, yet the vulnerability management ecosystem primarily reports observable manifestations of vulnerabilities rather than the underlying bugs [162, 88]. This creates a mismatch between the data typically available in public disclosures [128, 41], the information needed to evaluate vulnerability analysis tools, and the reasoning those tools are expected to perform. In this thesis, evaluation refers to comparing tool outputs against reconstructed reference information about a vulnerability, including the location of the bug, the location of the resulting error, and the characterization of the bug. The problem has three interconnected dimensions: a representational gap between manifestation and bug, practical constraints on reproducing vulnerabilities, and a resulting misalignment between what evaluation measures and what tools are expected to detect.

Representation gap: manifestation versus bug

Public vulnerability disclosures communicate observable consequences [161]—what fails and what impact it may have—because these elements support triage and response at scale. This representation, however, leaves unspecified which bug enabled the vulnerability and where that bug is located in code. MITRE's guidance recognizes this tension: *CVE descriptions emphasize impacts and attack vectors rather than weaknesses, and root cause mapping is not done accurately at scale by the vulnerability management ecosystem* [162]. As a result, research and tool evaluation often focus on the manifestation of a vulnerability rather than the bug that enables it [69, 57, 231, 177].

A concrete case illustrates the limitation. CVE-2020-8899 is described as a heap-based buffer overflow in Samsung's Android Operating System (OS) image codec stack, triggerable by a crafted MMS and leading to remote code execution. This description is operationally useful, but it primarily states *what happens* and *what an attacker can achieve*. The underlying bug—a missing verification of image body width and height during header parsing—is not explicit, even though the missing check allows later decoding under invalid size assumptions. The observable failure is therefore the end of a chain whose beginning is a bug in validation logic. Such a mismatch reflects the gap between reporting practice and reasoning about the bug in code.

Conceptually, vulnerabilities unfold from a bug to an error state when the bug is activated, and then to a failure that becomes observable and security-relevant [10, 103, 16]. Public datasets and benchmarks, however, rarely preserve this progression in a form that supports reconstruction and evaluation. In this thesis, *root cause* refers to the underlying reason the bug exists [162, 106, 16], expressed through a structured representation. *Ground truth* refers to the reconstructed reference information used for evaluation, including where the bug is located, where the resulting error is observed, and how the bug is characterized. When these elements are absent or collapsed into a single label [238], evaluation cannot distinguish whether a tool identifies the bug itself or only its manifestation.

Data and reproducibility constraints

Even when the root cause is conceptually clear, converting a vulnerability into an evaluation-ready instance is often infeasible [168]. Reports may be incomplete [88, 41], repository linkages unreliable [198, 45], and patches loosely referenced [245, 171] or entangled with unrelated changes, complicating bug localization [102]. These issues are compounded by ecosystem drift: builds break, dependencies disappear, toolchains evolve, and historical versions become difficult to reconstruct [168, 154, 222]. As a result, a vulnerability well documented in public disclosures may nonetheless be unusable for evaluation because the artifacts required for reconstruction and deterministic execution are missing or irrecoverable.

This exposes a structural fault line in empirical vulnerability research. On the one hand, reconstruction presupposes the ability to recover the bug–error–failure chain from available artifacts [20, 107]. On the other hand, empirical evaluation depends on executable instances that can be built, exercised, and replayed under controlled conditions [95, 168, 46]. When these requirements cannot be jointly satisfied [154], vulnerabilities differ not only in their semantic clarity but also in what forms of analysis and evaluation they can meaningfully support.

In practice, these distinctions are often treated implicitly. Vulnerabilities that support reconstruction, those that support reproduction, and those that support neither are frequently aggregated under a single notion of ground truth [14, 46, 154]. This conflation obscures the relationships among available artifacts, reproducibility, and evaluation validity, and underscores the need for benchmarks that make these constraints explicit rather than absorbing them as unexamined assumptions.

Evaluation misalignment and its implications

This representational gap matters directly for tools that aim to detect or localize vulnerabilities. Static analysis tools do not detect "remote code execution" as such; they issue warnings about code patterns, program states, or violated assumptions that may indicate a bug [179, 203, 83]. Evaluating such warnings, therefore, requires a clear reference for what counts as a correct result. If that reference is tied only to the observed failure or where a crash occurs, a tool may appear effective because it warns near the point of failure, even if it does not identify the underlying bug [8, 73, 186, 140]. For this reason, empirical studies must explicitly state what they treat as correct.

Existing datasets and benchmarks often preserve only part of the information needed for this distinction. Modern fuzzing infrastructures can efficiently generate reproducible failures [202, 154], but they do not systematically map crashes back to the code locations where the bug originates [15, 94, 107]. Conversely, studies on crash-driven root-cause localization often work on ad hoc subsets that do not serve as reusable benchmarks [15, 207, 53, 183]. Other benchmarks derived from real vulnerabilities in open-source software provide only coarse vulnerability locations, such as the affected function, which makes it difficult to tell whether a tool detects the bug or merely reports code near the propagated error [43, 140, 30, 258].

This mismatch affects more than measured accuracy. If evaluation rewards warnings that are merely close to the observed failure, tools are encouraged to report symptoms rather than causes. Such warnings may look correct under ground truth defined at the level of the manifestation, yet still offer limited help to developers trying to understand the bug and decide how to fix the code [43, 30, 94]. In this way, ambiguity in evaluation criteria propagates into ambiguity in tool outputs, linking benchmark design to warning usefulness and triage effort in practice [96, 120, 94].

1.2 Technical and Conceptual Background

This thesis relies on a small set of concepts for describing how vulnerabilities are reported, how they are represented in public data sources, and how their underlying bugs can be reconstructed from software artifacts. The goal of this section is not to survey the full landscape of vulnerability taxonomies and datasets, but to introduce the minimum background needed to understand the representational gap identified in Section 1.1 and the thesis choices that follow.

Vulnerability disclosure and metadata sources. The vulnerability disclosure process begins when a flaw is identified and reported through development or security channels, after which vendors or maintainers may publish advisories describing affected software and remediation. The Common Vulnerabilities and Exposures (CVE) Program [38] assigns unique identifiers to publicly disclosed vulnerabilities through authorized *CVE Numbering Authorities* (CNAs) [36], making CVE records the canonical identifiers used across the ecosystem. The National Vulnerability Database (NVD) [175] enriches CVE records with structured metadata that supports interoperability across security tools. In open-source ecosystems, databases such as Open Source Vulnerability (OSV) [80] and *GHSA* [75] complement this layer by linking advisories to repositories, commits, and affected version ranges. Automated fuzzing infrastructures such as OSS-Fuzz [82] provide further metadata by producing inputs, crashes, and sanitizer logs that capture how vulnerabilities manifest at runtime. Together, these systems provide complementary artifacts about vulnerabilities: identifiers and public descriptions from CVE, metadata from NVD, code history from OSV and GHSA, and reproduction from fuzzing.

Vulnerability standards and representational ambiguity. Despite coordinated identification and standardized reporting mechanisms, the content of CVE records remains uneven in completeness and structure. Although reporting templates encourage inclusion of *vulnerability type*, *root cause*, *impact*, *attacker type*, and *attack vector* [161], these details are usually conveyed through concise free-text descriptions whose technical language makes structuring prone to ambiguity and variability [133, 90, 7, 248]. In practice, disclosures emphasize observable consequences and exploitation outcomes, while the underlying cause is often omitted or only ambiguously described [88]. Structured metadata captures complementary aspects through standards such

as Common Vulnerability Scoring System (CVSS) [155], which describes severity, Common Platform Enumeration (CPE) [227], which identifies affected products, and Common Weakness Enumeration (CWE) [35], which enumerates types of software weaknesses. Because CWE mappings are commonly derived from CVE descriptions, they can inherit the same bias toward consequence and impact rather than the underlying issue that enabled the vulnerability.

Vulnerability causality and the propagation chain. This limitation motivates a representation of vulnerabilities that captures more than their observable outcome. Such a representation is made explicit by Avizienis *et al.* through the *fault* $\rightarrow$ *error* $\rightarrow$ *failure* chain [10]. In that model, a *failure* is the externally observable consequence of an incorrect state, an *error* is the internal incorrect state, and a *fault* is its hypothesized cause. This representation is important for understanding how vulnerabilities propagate. Still, the notion of *fault* remains intentionally abstract and, by itself, does not provide a concrete representation of the underlying cause. The Bugs Framework (BF) makes that structure more explicit at the code level by representing a weakness through *cause*, *operation*, and *consequence* [16]. In BF, the *cause* is either a *bug*, denoting a defect in code or specification, or a *fault*, denoting an erroneous operand such as an address or size. The *consequence* is either an *error*, which propagates as a *fault* to the next weakness, or a *final error*, which enables a *failure* (the failure classes used in this thesis are defined in Appendix A.2.3). A vulnerability is then a chain of such weaknesses, and the level of analysis in BF is at both the code and runtime levels.

Root cause and manifestation. The relevant level of analysis in this thesis is the improper code construct that enables the vulnerability, not the intermediate states through which it propagates. This choice aligns with static analysis tools, which generally operate on source-code abstractions rather than on runtime program states. Accordingly, we retain the $\langle cause, operation \rangle$ pair as a structured characterization of the code bug, which we refer to as the *root cause*, in line with the view in BF and CWE that the root cause corresponds to the first weakness in a vulnerability chain. In this representation, *operation* denotes the relevant program action[1], and *cause* denotes the qualifier of that action. The downstream *error* and resulting *failure* are treated separately as the *manifestation*. On that basis, we adopt a simplified representation of the vulnerability as a *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chain, where *error* refers to the final error state before the resulting failure, which corresponds to the *final error* in BF. Unlike the BF chain of weaknesses, our simplified chain does not model the intermediate propagation between the bug and the final error.

Ground truth, oracles, and reference locations. Any empirical evaluation requires an *oracle*: a specified criterion for deciding whether a technique matches the vulnerability of interest. In vulnerability studies, the oracle is often implicit, for example, by relying on file- or function-level proximity to crash context, which can conflate detection of the bug with proximity to its observed consequence. In this thesis, *ground truth* refers to the reconstructed reference information used by

[1] The operation labels come from the BF operation taxonomy, which we interpret at the code level (Appendix A.2.1).

an oracle, including the bug location, the error location, and the *root cause* characterization of the bug. These reference locations correspond to different points in the causal progression. The *bug location* is the code location associated with the bug, typically localized by patch artifacts, whereas the *error location* is the code location where the error, i.e., the final error defined above, is observed at runtime, typically localized by crash or sanitizer artifacts. These locations diverge, but they can also coincide when code is missing. A reliable evaluation must therefore state explicitly which reference location it targets.

SAST tools, warnings, and actionability. Static Application Security Testing (Static Application Security Testing (SAST)) [179] tools analyze source code without execution and report potential security issues as *warnings*, typically including a source location, a diagnostic message, a rule identifier, and often a mapped weakness category such as a CWE. In empirical evaluation, these warnings can be considered at two levels: *detection*, which asks whether a tool flags the relevant code location, and *assessment*, which asks whether the warning correctly characterizes the issue at that location. In practice, however, even a correctly localized warning may still be of limited use if it does not help distinguish an important bug from incidental or misleading signals. For this reason, SAST outputs are also studied in terms of *actionability*, that is, whether a warning provides information that is useful for deciding that the flagged code should be repaired. This distinction is important because real-world tools can emit very large numbers of warnings, many of which are ignored as *unactionable*, making triage a central part of effective static analysis.

1.3 Research Trajectory and Problem Refinement

This thesis originated from an initial plan to improve Automatic Program Repair (APR) for security vulnerabilities by integrating vulnerability-detection signals to improve localization and reduce wasted patch-validation effort. In the proposal, vulnerabilities were largely treated as aggregates of attributes mined from public sources, such as language, component, software type, and manifestation labels, and the profiling and benchmarking tasks were framed mainly as enabling steps toward comparative evaluations of SAST tools, Machine Learning (ML)-based vulnerability detection methods, and APR tools.

Two empirical realizations reshaped the direction of the work. First, during early experiments with SAST tools, we observed that evaluation depends strongly on which location is treated as ground truth, because the *bug location* identified in the patch often diverged from the *error location* observed in the crash. This matters because a SAST tool may identify the error location, yet directing an APR tool to fix it may only mitigate the vulnerability rather than resolve it, which requires fixing the bug. Second, as bug locations were reconstructed from the patch, many cases were better characterized as *missing* enforcement, such as absent checks, validation, or consistency constraints, rather than *incorrect* implementation. This matters because bugs involving missing code are intrinsically difficult to localize: what is omitted must often be inferred from

specification knowledge and developer intent that are not explicit in the code, which complicates manual annotation and makes automated detection and repair especially difficult.

These findings challenged the proposal hypothesis that stronger detection signals would translate into better APR localization and raised a more fundamental question: whether tools can report the relevant location at all when that location may not be explicitly represented in the vulnerable code. The thesis, therefore, shifted from *integrating the outputs of SAST tools into repair* to *evaluating whether SAST tools can identify repair-relevant vulnerability locations*. This reorientation led to three concrete tasks: (i) bug profiling grounded in patch artifacts, (ii) a reproducible benchmark that records both bug and error locations, and (iii) evaluation of SAST tools with ML-based triage under ground truth defined at the level of the bug.

1.4 Thesis Objectives and Research Questions

This thesis establishes the empirical and methodological basis for evaluating detection and repair techniques on the bug rather than its manifestation. For that, we study security bugs in C/C++ open-source systems, primarily memory-safety vulnerabilities associated with publicly disclosed CVEs, because this setting provides concrete artifacts for reconstructing the bug–error–failure chain, using fuzzing to reproduce the failure and patches to recover the bug. We therefore argue that evaluation should reflect this bug–error–failure propagation rather than rely solely on agreement with the observed manifestation. The research questions are organized accordingly: **RQ1** characterizes bugs in relation to their software context; **RQ2** determines what information a benchmark must record to represent and reproduce that propagation for evaluation; and **RQ3** assesses whether outputs from SAST tools, augmented with ML-based triage, can be made actionable under ground truth defined at the level of the bug.

Profiling security bugs. Current vulnerability datasets and evaluation practices largely describe vulnerabilities in terms of manifestations (e.g., crash types, CWE labels, or failure symptoms), while the underlying bugs and their software context remain weakly represented. Consequently, it remains unclear whether real-world security bugs exhibit stable and interpretable patterns or whether superficially similar failures arise from different bugs, motivating the following question:

RQ$_1$. *What root causes and contextual software factors characterize real-world security bugs, and do stable, interpretable profiles emerge across cases?* gm We reconstruct vulnerabilities at the level of the bug from patch artifacts and associated metadata, distinguishing *Missing* versus *Incorrect* causes [2] and recording the surrounding software context in which bugs occur. Each reconstructed bug is linked to the resulting runtime errors and externally observable failures, allowing profiles to be expressed as propagation chains rather than clusters of symptoms. This enables comparison of how different bugs lead to similar security consequences and how similar bugs diverge in propagation depending on context.

[2] While the *Missing* and *Incorrect* distinction resembles BF's *Missing Code* and *Erroneous Code* distinction, we retain the CWE glossary terms as the reference (see Appendix A.2.1).

Establishing these profiles defines the unit of analysis used throughout the thesis. A normalized representation of bugs and their propagation is necessary to design benchmarks that capture the distinction between root cause and manifestation (RQ2) and to interpret tool outputs in terms of repair-relevant localization rather than incidental agreement with failure context (RQ3).

Benchmark requirements for reliable evaluation. Existing benchmarks typically encode labels and locations that reflect the manifestation of a vulnerability, conflating the underlying bug with its runtime consequences. As a result, evaluations based on such benchmarks may reflect only incidental agreement with failure context rather than the real ability of SAST tools to identify the bug. Reliable evaluation requires a benchmark that captures the structure of the vulnerability, leading to the following question:

RQ$_2$. *Which attributes must a vulnerability benchmark record to enable reliable evaluation?*

We frame benchmark construction around the bug profiles established in RQ1, asking what information must be recorded so that evaluation distinguishes between the bug and the resulting error. This requires executable cases that can be triggered to reproduce the failure while remaining linked to the patch artifacts used to reconstruct the bug. The benchmark records the bug location, the error location, and the artifacts to reproduce the failure and interpret their relationship.

Defining these requirements operationalizes the unit identified in RQ1 into a reproducible basis for evaluation. Without such representation, tool assessment cannot distinguish bug detection from the detection of its propagated manifestations, and evaluating whether outputs from SAST tools, augmented with ML-based triage, are actionable (RQ3) would lack a sound ground-truth model.

Evaluation of SAST outputs and actionability. Conventional evaluation of SAST tools often treats detection as a binary matching problem against vulnerability labels or reported locations, without distinguishing whether a tool identifies the bug or only aligns with the propagated errors. This obscures whether reported findings are actionable for repair, since remediation depends on locating and understanding the bug that must be fixed rather than its visible consequences. With ground truth that distinguishes the bug from the resulting error, evaluation can measure what a tool actually identifies, leading to the following question:

RQ$_3$. *To what extent are outputs from SAST tools with ML-based triage actionable under an evaluation that distinguishes the bug from the resulting error?*

We evaluate SAST tools under two complementary oracles: detection at the *bug location*, corresponding to the root cause that must be repaired, and detection at the *error location*, corresponding to where the manifestation becomes visible. Beyond localization, we assess whether the SAST tool findings indicate that the detected condition corresponds to the root cause, or whether apparent matches are incidental. Building on this, we investigate whether explanation-based ML triage can reduce unactionable warnings while preserving findings that align with bug location, thereby improving the practical utility of tool outputs within the defined scope.

This evaluation builds directly on the previous questions: it uses the bug profiles characterized in RQ1 and the benchmark established in RQ2, which records both bug and error locations. Together, these make it possible to assess not only detection performance but also whether SAST tools identify the root cause that must be repaired.

In summary, the investigation establishes three central findings corresponding to the research questions. First, reconstructed vulnerabilities reveal interpretable bug profiles characterized by missing or incorrect causes within specific software contexts. Second, reliable evaluation requires benchmarks that separate the bug from its manifestation, which conventional benchmarks often collapse into a single location. Third, empirical evaluation shows that SAST tools often align with manifestation locations rather than underlying bugs, but explanation-based ML triage can substantially reduce unactionable warnings while preserving actionable warnings.

1.5 Research Approach and Methodological Strategy

To address the thesis objectives and research questions, the methodological approach builds on the vulnerability structure introduced earlier. The approach reconstructs bugs from carefully selected CVEs using consistent metadata, linked patches, and sufficiently informative descriptions, and represents them in a structured, machine-usable form that distinguishes the bug from its manifestation. This vulnerability structure is used to evaluate analysis tools based on their ability to localize the bug. The root-cause representation is then used to normalize tool findings for assessment and triage: the operation supports comparison against the reconstructed bug, while the combination of cause and operation is used to generate natural-language explanations of proper or improper program actions. These explanations are then used to train ML models to filter unactionable warnings. Across the thesis, reconstruction, representation, and evaluation form a cumulative progression in which each component provides the basis for the next.

Bug profiling. Vulnerabilities are reconstructed at the level of the bug using CVE records as entry points, but only after cases satisfy strict eligibility and metadata criteria so that each instance is semantically consistent and traceable to concrete artifacts. To retain application-level vulnerabilities and resolve affected products to accessible open-source repositories, we use CPE metadata, which provides the product identifiers needed for filtering and linkage. Repository and commit references, together with external package identifiers, are then used to link cases to code changes in open-source, restricting analysis to contexts where the bug can be directly inspected. Additional constraints on exploitability and scope limit the corpus to vulnerabilities whose activation depends primarily on intrinsic program behavior and whose effects remain within a single authorization boundary, enabling reconstruction of closed bug–error–failure chains. Patch diffs are used to reconstruct the bug and localize its position in the code, while vulnerability descriptions are used to reconstruct the manifestation from the consequence and impact; metadata serves as linkage and contextual support rather than as ground truth. The reconstructed information is

then mapped to vulnerability-chain attributes and software-context attributes. The former record the *root cause*, *error*, and *failure*, while the latter capture the *software type*, the *functional area* surrounding the bug, and the *programming language* of the affected file, using sufficiently coarse categories to reduce reporting-style variation and support comparison across cases.

Benchmark crafting. To build the benchmark, we apply the bug reconstruction procedure to a separate set of candidate vulnerabilities drawn from fuzzing disclosures (OSV records and OSS-Fuzz issue reports), which provide reproducer inputs and crash artifacts. Candidate cases are linked to CVEs through multi-signal matching and historical version analysis, followed by build reconstruction, containerized environment assembly, and deterministic crash validation. Each validated instance records two locations: the bug location grounded in the patch and the error location grounded in the crash, preventing evaluation from conflating proximity to the manifestation with identification of the bug. Only reproducible cases are included in the benchmark core, with artifact packaging and version information ensuring traceability, resulting in a corpus that prioritizes vulnerability structure over breadth.

Tool evaluation and actionability. A set of build-independent C/C++ SAST tools, selected from the literature for default rule coverage and Static Analysis Results Interchange Format (SARIF)-compatible output, is evaluated against the benchmark using two exact-line oracles, that is, ground-truth source-code locations defined at single-line granularity: detection at the bug location and detection at the error location. This exposes whether findings correspond to the bug itself or only to its manifestation context. Assessment of reported semantics maps tool labels, often given as coarse CWE categories, into operation categories and compares them with the operation identified in the root cause, providing a finer-grained notion of correctness than coarse CWE label agreement alone. A tool-agnostic ML triage layer then generates natural-language explanations for findings using the root cause structure and trains a classifier to filter unactionable warnings under supervision aligned with that characterization. Variability in tool reporting and explanation generation is treated explicitly in the analysis.

1.6 Summary of Contributions

Table 1.1 summarizes the main contributions of the evaluation framework introduced in this thesis. The framework treats security bugs as the central unit of analysis, models their relationship to resulting errors and failures, and evaluates tool outputs — including ML-assisted triage — using a benchmark that distinguishes the bug from the error.

RQ1: Profiling security bugs. This contribution establishes the root cause as the central unit of analysis through a reconstruction process that links CVE records to patch artifacts and recovers *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chains as the structured representation of vulnerabilities. The process

RQ	Contribution	Role in Framework	Type
RQ1	Automated pipeline that combines metadata and patch diffs to identify cases suitable for reconstruction	Defines bugs and software context as the basis for analysis	Artifact/ Empirical Finding
	Empirical identification of dominant bug profiles and their manifestation chains	Establishes which bugs recur and how they lead to errors and failures	Empirical Finding
RQ2	Reproducible benchmark with bug and error locations	Records separation between cause and manifestation for analysis and evaluation	Artifact
	Evidence of localization divergence & manifestation semantic compression	Show how evaluation at the manifestation level can conflate the bug with the error	Empirical Finding
	Design principles for reliable vulnerability benchmarks	Specifies the attributes a benchmark must record to capture vulnerability structure	Method. Insight
RQ3	Evaluation of SAST tools at bug and error locations	Measures whether tools identify the bug or only align with the resulting error	Method. Framework
	Assessment of tool semantics at the level of operation	Tests whether tool-reported labels correspond with the operation in the root cause	Empirical Finding
	Tool-agnostic explanation layer and ML-based filter for actionable warnings	Transforms warnings into causal explanations & reduces non-actionable warnings	Artifact

Table 1.1: Contributions organized by their role in the evaluation framework.

yields **55** high-confidence reconstructions from **66** candidate vulnerabilities and supports a representation that separates, normalizes, and disambiguates the bug, its manifestation, and its software context. Empirical analysis reveals a small set of dominant security bug profiles alongside various infrequent, context-specific bugs and shows that identical high-severity outcomes—mainly *code execution*—arise through distinct error-propagation paths depending on whether the bug reflects *missing* or *incorrect* enforcement.

RQ2: Benchmark design for reliable evaluation. Building on the vulnerability structure established in RQ1, this contribution translates that representation into benchmark form by reproducing a distinct subset of vulnerabilities from OSV and NVD disclosures, yielding an executable core of **30** manually validated, crash-reproducing cases derived from **69** OSV–CVE links and **40** buildable candidates. Each instance is bound to a specific vulnerable version and records two locations: the bug location and the error location. Empirical analysis shows that these locations frequently diverge across files and functions, which means localization must be evaluated against more than one ground-truth target. Labels tied only to the manifestation location fail to capture the root cause, which often differs across bugs that lead to the same outcome. From these observations, the work derives the requirements for reliable evaluation: explicit recording of bug and error locations, root-cause and manifestation labels grounded in fix and crash evidence. Supporting artifacts are the vulnerable program version, the crashing input, and the corresponding patch.

RQ3: Evaluation of SAST tools and triage. This contribution evaluates **four** SAST tools under a protocol that distinguishes detection at the bug location from detection at the error location, making it possible to measure whether tools identify the bug or only align with the resulting error. It also assesses whether those detections are meaningful by mapping reported CWEs into

operation labels and comparing them with the operation identified in the root cause. A tool-agnostic reasoning layer generates natural-language explanations for heterogeneous warnings and attaches them to SARIF outputs, enabling interpretable detection. A lightweight classifier trained on **135** explanations filters unactionable warnings, reducing false positives while preserving repair-relevant true positives within the evaluated scope.

Artifact availability. A shared data-acquisition backbone supports all stages: `nvdutils`[3] provides structured access to NVD records, and `github-lib`[4] supports repository and commit retrieval. For *RQ1*, `pydantic-cwe`[5] supplies typed CWE abstractions, `cve-driller`[6] extracts vulnerability concepts, and `cpelib`[7] supports structured CPE handling. For *RQ2*, `osv-utils`[8] and `osv-reproducer`[9] provide OSV interaction and OSS-Fuzz reproduction tooling for version-anchored, crash-validated benchmark instances. For *RQ3*, containerized SAST tool images[10] ensure reproducible analysis environments. Complementary artifacts—including datasets, benchmark instances, evaluation logs, pipelines, and the trained triage model—are prepared for release[11] to complete the reproducibility chain.

1.7 Structure of the Thesis

The thesis comprises three main chapters beyond this introduction and conclusion, progressing from vulnerability representation to benchmark construction and tool evaluation. Each chapter addresses one stage of the research design and corresponds directly to the research questions defined in Section 1.4.

Chapter 2: Profiling Security Bugs in Open Source Software (OSS) (RQ1). Chapter 2 answers RQ1 by showing that vulnerabilities can be represented as *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chains centered on the root cause, and that this representation exposes bug profiles shaped by the program structure. To do so, it reconstructs vulnerability instances and recovers the root cause, its manifestation, and its software context to identify bug patterns and their characteristic error pathways.

Chapter 3: Crafting a Vulnerability Benchmark (RQ2). Chapter 3 answers RQ2 by showing that reliable evaluation requires capturing both the bug and error locations, together with labels grounded in patch and crash evidence. Building on the vulnerability structure established in Chapter 2, it reproduces vulnerabilities as executable, version-bounded instances and records these two ground-truth locations for benchmark use.

[3] https://pypi.org/project/nvdutils/
[4] https://pypi.org/project/github-lib/
[5] https://pypi.org/project/pydantic-cwe/
[6] https://pypi.org/project/cve-driller/
[7] https://pypi.org/project/cpelib/
[8] https://pypi.org/project/osv-utils/
[9] https://pypi.org/project/osv-reproducer/
[10] https://hub.docker.com/repository/docker/epicosy/thesis-sast/tags
[11] https://github.com/epicosy/phd_thesis

Chapter 4: Evaluating SAST Tools and ML-Based Triage (RQ3). Chapter 4 answers RQ3 by showing that static analysis tools align more often with error locations than with bug locations, and that warning actionability improves when triage reasons about the flagged code in terms of root cause. Using the benchmark from Chapter 3, it measures detection performance at both locations, assesses whether warnings correctly characterize the operation at the detected location, and introduces an ML-based triage layer that generates structured explanations to distinguish actionable from unactionable findings.

Chapter 5: Conclusions and Outlook. The final chapter synthesizes the answers to RQ1–RQ3, discusses methodological limitations and validity, and outlines implications for vulnerability analysis, evaluation practice, and security research. It also identifies directions for extending the benchmark and its use in tool assessment beyond the current corpus and tooling scope.

1.8 Relationship to Prior Publications

This thesis is an integrated monograph. While selected chapters extend prior peer-reviewed publications, none of the chapters are reproduced verbatim, and all datasets, benchmark instances, reconstruction procedures, and the vulnerability representation presented here were developed specifically for this dissertation. Where applicable, this section clarifies how prior work served as a methodological foundation and delineates the novel contributions introduced in each chapter.

Chapter 2. Chapter 2 is informed by prior work examining how vulnerability representation and label quality affect downstream analysis and learning. One line of work informed the software-context axis used in profiling, including software-type classification using language models [59]. Another produced the TENET framework [192], which emphasized reproducible pipelines for vulnerability detection and highlighted a key obstacle for Root-Cause Analysis (RCA): automated labeling from patch diffs and NVD metadata is often too noisy to reliably capture the bug, since real-world patches may include refactoring, tests, or bundled fixes, and public labels emphasize manifestations rather than the underlying cause. Engagement with the NIST's BF and BF-based CVE labeling [21, 17] further reinforced the value of bug annotation. Synthesizing these lessons, Chapter 2 focuses on reconstructing security bugs from verifiable open-source evidence and organizing them into unified profiles with explicit *bug* $\rightarrow$ *error* $\rightarrow$ *failure* continuity; these profiles provide the conceptual basis for subsequent benchmark design and evaluation.

Chapter 3. Chapter 3 extends the published paper *Maestro: A Platform for Benchmarking Automatic Program Repair Tools on Software Vulnerabilities* at the *31st International Symposium on Software Testing and Analysis (ISSTA)* 2022 [190]. MAESTRO introduced a decentralized architecture for reproducible APR experimentation and demonstrated its use on curated datasets, while identifying the scarcity of reproducible benchmarks derived from real-world vulnerability disclosures as a central limitation. In this dissertation, that limitation is addressed through a systematic

methodology for transforming fuzzing-based disclosures and public version-control artifacts into executable, commit-linked benchmark instances compatible with MAESTRO's schema (repository linkage, build specification, vulnerable commit anchoring, vulnerability identifiers, and precise location metadata). These benchmark instances record both the bug and error locations. The benchmark construction and vulnerability reproduction procedures were not part of the original MAESTRO publication and are unique to this dissertation.

Chapter 4. Chapter 4 extends the published paper *Tenet: A Flexible Framework for Machine-Learning-based Vulnerability Detection* at the *2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)* 2023 [192]. TENET introduced a modular ML framework for vulnerability detection that labeled code fragments as vulnerable or safe using diff- and static-analysis-based mechanisms. It remained limited to vulnerability detection pipelines and identified improvements to the labeling process as future work. In this dissertation, those labeling strategies are extended to the triage of static-analysis warnings. Chapter 4 introduces a hybrid strategy that combines diff-derived vulnerable and fixed locations with static-analysis outputs computed on both code versions, enabling supervision at the level of individual SAST warnings. This reformulates vulnerability detection as actionable-warning discrimination over the benchmark instances introduced in Chapter 3. The integration of root-cause grounding and structured warning explanations strengthens labeling rigor and extends TENET's methodology beyond its original scope. These elements were not part of the original TENET publication and are unique to this dissertation.

Authorship context. I am the primary author of the publications underlying Chapters 3 and 4, including [190, 192]. The dissertation advances these works through new benchmark construction, root-cause reconstruction, integration into SAST triage, and SAST tool evaluation that accounts for the separation between bugs and their manifestation. I am also the primary author of *Evaluating Deep Neural Networks in Deployment (A Replicability Study)* (ISSTA 2024) [191], which addresses evaluation and replicability of tools that assess the reliability of deep neural networks.

Related publications. The following co-authored publications align with the broader research agenda of this dissertation and contributed additional methodological insights. They are cited within individual chapters where relevant and are not directly extended in this thesis:

- *A transformer-based ide plugin for vulnerability detection* at the *37th International Conference on Automated Software Engineering (ASE)* 2022 [147];
- *Exploring transformers for multi-label classification of java vulnerabilities* at the *22nd International Conference on Software Quality, Reliability and Security* 2022 [148];
- *A study of fine-tuned language models in vulnerability classification* at *12th International Symposium on Digital Forensics and Security (ISDFS)* 2024 [59];
- *Exploring AI for vulnerability detection and repair* at the *Cyber Awareness and Research Symposium (CARS)* 2024 [60];

Chapter 2

Profiling Security Bugs in OSS

2.1 Introduction

This chapter addresses the disconnect between common vulnerability reporting practices and the root-cause understanding needed for detection and patching. It presents an approach for reconstructing the vulnerability chain from vulnerability reports and patches, identifying the bug that enables the vulnerability, the resulting error, and the final failure. Using a curated corpus of vulnerabilities with sufficient information for reconstruction and labeling, the chapter identifies recurring combinations of root cause across software contexts as security bug profiles, and examines how these profiles propagate into errors and failures.

2.1.1 Motivation and gap

Major vulnerability reporting channels—CVE records [38] and the NVD catalog [195]—have historically described vulnerabilities primarily through their *manifestations*, such as buffer overflows, injections, and exposures. This emphasis is practical because manifestations are externally observable failure modes that cross protection boundaries and therefore demand immediate response. However, describing vulnerabilities in terms of their manifestations captures only the externally visible failure state and does not explain the underlying root cause. Even when vulnerability records link patches or source revisions, those artifacts do not directly identify that root cause, and recovering it still requires substantial interpretation. As a result, public vulnerability records provide only a limited root-cause signal.

This emphasis on manifestations is reinforced by how vulnerabilities are represented in current standards and classifications. CWE, the dominant vocabulary for vulnerability classification, often mixes the underlying cause with its downstream consequence in a single weakness label, such as "Improper Neutralization" alongside "SQL Injection," making the relevant elements of the vulnerability difficult to isolate. MITRE's Root Cause Mapping guidance [162] acknowledges this problem by recommending that the assigned CWE describe the issue that led to the vulnerability,

that is, its first weakness in the chain. However, CWE does not consistently express the root cause, because weaknesses do not always correspond to self-contained elements in the causal structure of a vulnerability. The NIST's BF [19, 16] makes that causal structure more explicit by modeling vulnerabilities in terms of bugs, errors, and failures, yet its basic unit still combines cause and consequence within a single weakness instance. As a result, current vulnerability representations still mix the improper code construct that enables a vulnerability with the erroneous runtime states through which it propagates.

To separate these elements, this chapter removes the consequence from the weakness representation and reduces the unit of analysis to the improper code construct, that is, the bug that constitutes the root cause of the vulnerability. This focus matters for detection because security analysis tools—particularly SAST—are designed to identify bugs in code, not the runtime manifestations they eventually produce. Since the resulting errors may propagate far from the bug, and even across software components, understanding vulnerabilities through their root cause helps clarify what such tools should detect. It also motivates examining the surrounding software context, which may help explain why certain bug patterns recur in some implementation settings and, in turn, where analysis and defenses are likely to be most effective.

2.1.2 Bug profiling approach and scope

Under this perspective, we represent each reconstructed vulnerability through three complementary elements: *root cause*, *software context*, and *manifestation*. Root cause characterizes the bug that enables the vulnerability through its *cause* (Missing vs. Incorrect) and the *operation* in which it occurs. Software context records the *software type*, *functional area* of the defective code region, and *programming language*, while manifestation captures the observed *error* and resulting *failure*. In this schema, *root cause* and *software context* form the basis for later bug profiling, enabling recurring bug patterns to be analyzed across software areas.

The analysis begins from the reported manifestation details in the vulnerability description and examines the corresponding patch to identify the code change that fixes the underlying bug. In this way, each case is reconstructed as a vulnerability chain linking the bug constituting the root cause of the vulnerability to the resulting *error* and final *failure*. This process mirrors the reasoning typically used during debugging: starting from an observed failure, backtracking errors, and examining the relevant code until the bug is identified. While the recovery of vulnerability artifacts is automated, bug localization and the labeling of *root cause*, *software context*, *error*, and *failure* are performed manually.

Our analysis focuses on *software* vulnerabilities that can be reconstructed and compared, prioritizing fidelity over breadth. We retain cases with *stable* and *consistent* metadata, whose activation depends on few conditions beyond reaching the vulnerable code, and whose impact is *critical* while remaining within the affected component, keeping the resulting set comparable. Since recovering the underlying bug requires code evidence, the usable corpus is limited to *open-source projects* with traceable repositories and accessible patches. Our scope is not to assemble the largest possible corpus, but to retain cases that support reliable reconstruction; this, in turn, makes visible

which metadata problems and missing artifacts prevent many vulnerabilities from being retained for analysis.

2.1.3 Research questions and contributions

Within this curated corpus, the reconstructed cases enable analysis of recurring bugs across software contexts and the manifestations they produce, motivating the following research question:
RQ1. What are the profiles of software security bugs and their manifestations?

> Specifically, we ask whether security bugs that enable real-world vulnerabilities exhibit interpretable patterns when characterized by *cause* and *operation* across *software context*, and whether those patterns differ. We also examine how these patterns relate to the resulting *errors* and *failures* through which bugs propagate.

To answer this question, the chapter contributes the following artifacts and empirical findings:

- **A vulnerability reconstruction workflow** that combines public artifacts to identify and reconstruct suitable cases, while leaving bug localization and labeling as manual steps.
- **A curated dataset of critical vulnerabilities** comprising 55 reconstructed and labeled cases represented through *root cause*, *software context*, and *manifestation*, which can serve as a benchmark for evaluating future automated and Artificial Intelligence (AI)-based methods.
- **An empirical characterization of security bugs across software context**, showing that *Missing Data Check* bugs in *Data Processing* areas of *PHP Web Applications* account for the largest share of cases, while *Data Type* mistakes in *C* form the main *Incorrect* profile.
- **An empirical characterization of how security bugs manifest**, showing that the main bug profiles frequently converge on *Code Execution* but reach it through distinct errors depending on whether the bug reflects missing or incorrect operations.
- **Implications for reporting, research, tooling, and practice.** We discuss how limitations in vulnerability metadata, description quality, and artifact availability constrain scalability and external validity; how improved reporting and AI-based methods could reduce these bottlenecks; how the resulting profiles provide an empirical basis for refining existing taxonomies; how *Missing* bugs expose a particularly difficult target for automation; and how recurring software hotspots can inform prioritization and hardening in practice.

Although the resulting corpus is small, this reflects the current state of vulnerability reporting: if metadata were more complete, descriptions more standardized, and fixing commits were widely available, bug profiling could scale to larger corpora and yield more generalizable insights. AI-based methods could help normalize and enrich vulnerability records and assist with the manual parts of reconstruction and labeling workflow, with the present dataset serving as a benchmark for evaluating how effectively such methods approximate expert judgment.

2.1.4 Publications context and roadmap

This chapter builds on prior work that explored how vulnerability *representation* and *label quality* affect downstream tasks. One line of work informed the *software context* axis used here, including *software type* classification using language models [59]. Another line produced the *Tenet* framework [192], which emphasized reproducible end-to-end pipelines for vulnerability detection and highlighted a key obstacle: automated labeling based on patch diffs and NVD metadata is often too noisy to localize the bug reliably, since real-world patches may include refactoring, tests, or bundled fixes.

In parallel, engagement with the NIST's BF and BF-based CVE labeling [21] reinforced the value of bug annotation, since CVE descriptions and NVD labels primarily describe manifestations rather than the underlying bug. Synthesizing these lessons, the present chapter focuses on reconstructing the bug that constitutes the root cause of a vulnerability from verifiable open-source evidence and organizing the resulting cases into profiles with explicit *bug* $\rightarrow$ *error* $\rightarrow$ *failure* continuity. These profiles are then used in subsequent chapters to assess whether static analysis tools target the root causes that most frequently enable real-world vulnerabilities.

Roadmap. Section 2.2 introduces background on vulnerability causality and manifestation, and Section 2.3 reviews related work on vulnerability classifications and taxonomies. Section 2.4 presents the reconstruction and profiling methodology. Section 2.5 reports on the identified bug profiles and manifestation chains, followed by implications and limitations in Section 2.6. Section 2.7 concludes the chapter.

2.2 Background

This section introduces the conceptual foundations required for analyzing vulnerability causality. Section 2.2.1 defines software vulnerabilities through the bug–error–failure chain, grounding the discussion in dependability theory. Section 2.2.2 situates vulnerabilities within the OSS lifecycle and ecosystem, including disclosure and remediation processes. Section 2.2.3 reviews representation standards and taxonomy frameworks that structure vulnerability knowledge. Section 2.2.4 surveys the databases and repositories that operationalize these standards and provide the empirical artifacts—advisories, commits, and version histories—used in vulnerability analysis.

2.2.1 Software Vulnerabilities and the Bug–Error–Failure Chain

The term *vulnerability* is defined differently across perspectives—standards bodies, taxonomies, and dependability theory—but all describe a system weakness that can be triggered to compromise its intended function. NIST [218] views a vulnerability as a weakness exploitable by a threat source, CWE [39] emphasizes weaknesses that enable unauthorized actions or data access, and CVE [37] frames it as a flaw whose exploitation degrades Confidentiality, Integrity, and Availability (CIA). Here, *confidentiality* denotes protection against unauthorized disclosure of information,

integrity denotes protection against unauthorized modification of programs or data, and *availability* denotes the degree to which a system or component remains operational and accessible when required [10, 103, 218].

From a dependability standpoint, Avizienis *et al.* [10] define a vulnerability as an *internal fault that enables an external fault to harm the system*, explicitly embedding causality. In this view, a vulnerability represents a fault—introduced during design, implementation, or operation—that may remain dormant until activated by certain inputs or states. Once triggered, the system transitions from a correct to an incorrect state (an *error*), which may propagate and become externally visible as a *failure*.

This fault–error–failure progression (Figure 2.1) describes the underlying manifestation chain common to both reliability and security failures. The IEEE Standard Glossary [103] characterizes a fault as an incorrect step or data definition, an error as the resulting incorrect state, and a failure as the system's inability to deliver required functionality. A vulnerability, therefore, can be understood as a *security-relevant fault*—one whose activation compromises CIA properties rather than merely functional correctness.

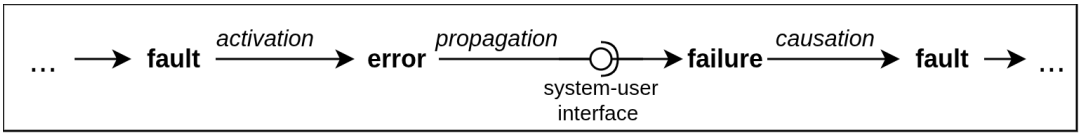


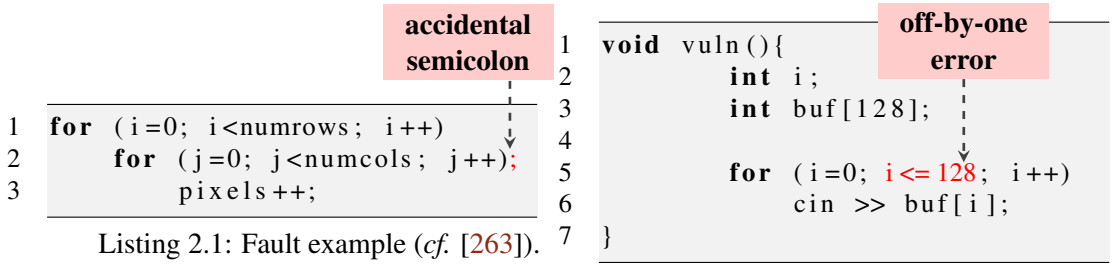
Figure 2.1: Manifestation chain of defect, error, and failure (adapted from [10]).

Not all faults lead to security consequences. *Functional faults* may cause benign errors, such as computation inaccuracies (e.g., Listing 2.1) or crashes, while *security faults* expose system assets to unauthorized influence or disclosure. These faults undermine one or more of the CIA attributes—availability (operational readiness), integrity (protection from unauthorized modification), and confidentiality (protection from unauthorized disclosure) [218, 103].

Figure 2.2 illustrates this distinction. In Listing 2.1, a logic fault prevents a variable from updating, producing an incorrect but non-threatening outcome. In Listing 2.2, a bounds-checking error in a loop enables a buffer over-read that can be exploited to execute arbitrary code—transforming a defect into a vulnerability. In essence, when a fault's activation leads to an exploitable error that violates CIA properties, it becomes a *manifested security vulnerability*.

2.2.2 Vulnerabilities Across the OSS Lifecycle and Ecosystems

Vulnerabilities emerge throughout the software lifecycle, with origins, timing, and locations varying. Landwehr *et al.* [126] characterize these dimensions as *genesis* (how a vulnerability arises), *time of introduction*, and *location*. This work focuses on vulnerabilities that arise from inadvertent developer mistakes during implementation. Such vulnerabilities are most often introduced in the **development phase**, within source code, where conceptual mistakes become concrete defects. Their *location* determines scope: vulnerabilities in privileged layers (e.g., operating systems or



Listing 2.1: Fault example (*cf.* [263]).

Listing 2.2: Security fault example (*cf.* [28]).

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

libraries) can escalate privileges or compromise system integrity, whereas application-level vulnerabilities typically expose data or disrupt functionality [119].

Modern software engineering amplifies these risks through pervasive **code reuse**. Open-source components and libraries, often included transitively, extend dependency chains that can propagate vulnerabilities across entire ecosystems [239, 78, 76]. As recent supply chain incidents demonstrate, a single compromised dependency can cascade through thousands of downstream systems [234, 152]. Mitigation thus depends on identifying vulnerable third-party components and understanding how they integrate within the broader system [221].

The **vulnerability lifecycle** models how such faults evolve from creation to mitigation. Lin *et al.* [137] describe six stages: generation, discovery, disclosure, patch release, patch application, and demise. Vulnerabilities progress from *unknown* to *disclosed*, moving through zero-day, one-day, and historical phases as patches are released and adopted. The associated risk peaks between disclosure and remediation, when exploitation potential is highest.

In OSS, this process is more dynamic and decentralized. Discovery, disclosure, and patching often occur concurrently and in public view. Early or uncoordinated disclosures compress response times, complicating patch tracking and vulnerability verification. Consequently, **vulnerability detection** and **patch detection** form a continuous feedback loop rather than discrete steps. The former identifies latent defects before exploitation; the latter validates whether released fixes resolve them. Together, they define the operational boundaries for analyzing how vulnerabilities manifest and propagate within open-source ecosystems.

2.2.3 Vulnerability Classification Frameworks and Bug Representation

Understanding how vulnerabilities are represented is central to this thesis, because different frameworks vary in how clearly they express causation. Early taxonomies, such as that of Landwehr *et al.* [126], offered an initial structure by classifying vulnerabilities according to their *origin*, *location*, and *time of introduction*. Subsequent approaches adopted terms such as *errors*, *flaws*, and *weaknesses*, reflecting different units and levels of analysis. Among them, CWE [35] became the dominant standard for organizing weaknesses, but its entries do not always cleanly isolate the underlying cause of a vulnerability. The BF provides a more explicit causal model that represents,

within a single weakness, both code-level elements (the bug and fault) and state-level elements (the error states). Against this background, this subsection introduces the *root cause* representation used in this thesis to characterize the bug separately from its downstream *manifestation*.

Common vulnerability standards. CWE provides a hierarchical schema for organizing weaknesses at different abstraction levels, from broad classes (e.g., improper input validation) to specific variants (e.g., improper validation of specified type of input). Each entry describes technical properties such as affected resources, common consequences, and related weaknesses, enabling consistent categorization of defect types. CWE operates within a broader representation ecosystem that includes the CVE catalog [48], which identifies individual vulnerability instances, the CVSS [155], which quantifies severity, and the CPE [227], which names affected products and versions. Together, these standards link weakness types, vulnerability instances, affected platforms, and impact metrics.

CWE and representational ambiguity. Within this ecosystem, CWE does not always isolate the underlying cause of a vulnerability cleanly. Many entries combine cause, resource, and consequence within a single weakness description [238], reflecting its historical development from PLOVER, a bottom-up effort that grouped recurring real-world vulnerabilities into weakness classes [34]. As a result, many CWE entries closely correspond to specific types of vulnerabilities, making their underlying elements difficult to isolate. This ambiguity is further compounded by CWE's hierarchical structure, where similar concepts may appear at different levels of abstraction [40]. Several works have attempted to recover a clearer causal structure from CWE-based descriptions. *Semantic Templates* (ST) reorganize CWE entries into structured components such as software faults, resources, characteristics, and consequences [238], while *Software Fault Patterns* (SFP) relate recurring code-level faults to their runtime manifestations [150]. These approaches make vulnerability structure more explicit, but because they still derive that structure from CWE weaknesses, they inherit some of the same ambiguity about how the underlying elements relate.

Bugs Framework and vulnerability modeling. To address this limitation, the BF makes the causal structure of weaknesses more explicit by representing a weakness as $\langle \textit{cause}, \textit{operation} \rangle \rightarrow \textit{consequence}$, where the *cause* is either a bug or a fault, the *operation* is the minimal input-process-output code that can produce or propagate improper data, and the *consequence* is the resulting *error* or *final error* [16]. In this model, a bug initiates a propagation chain, leading to errors that may become faults for subsequent weaknesses and culminating in a *final error* that enables a security failure. BF thus refines the bug–error–failure progression introduced earlier into a structured model of weakness propagation grounded in both program operations and program states during execution. It further organizes weaknesses into categories such as *Data Type*, *Input/Output (Data) Check*, and *Memory*, while aligning them with corresponding CWE entries for interoperability [19, 18]. By design, BF represents cause and consequence within each weakness, which lets it specify causation within a weakness and propagation between weaknesses.

Root cause as bug representation. This thesis focuses on the bug that initiates the chain, which motivates the representation adopted here. BF represents code-level and state-level elements within a weakness, through its *consequence* and through its notion of *cause*. In BF, *cause* may denote either an improper operation (*bug*) or an improper operand (*fault*), so the *cause* already spans both defective code and erroneous program state. When the cause is a bug, BF points to the first weakness in the vulnerability chain as the *root cause*, which aligns with what both CWE's guidance [162] and software-engineering terminology [106] describe as the underlying cause of the vulnerability. At that level, BF and CWE contain the same basic ingredients of a root-cause representation: a program action and a qualifier expressing what is wrong about that action. In BF, that qualifier may take values such as *Missing* or *Erroneous Code*; in CWE, a closely related distinction appears through qualifiers such as *Missing* and *Incorrect*. This thesis structures those two ingredients as the pair $\langle cause, operation \rangle$, following BF, and specializes it to code bugs to characterize only the bug that initiates the vulnerability. That representation defines the *root cause*, where *operation* denotes the relevant program action and *cause* denotes the qualifier of that action. The downstream *error* and resulting *failure* are treated separately as the *manifestation*. In this way, *consequence* is removed from the representation of the weakness itself, and the unit of analysis is narrowed to the improper code construct that security analysis tools are intended to detect.

2.2.4 Vulnerability Databases and Repositories

While standards define how vulnerabilities are classified and described, empirical analysis relies on databases that collect, enrich, and disseminate structured vulnerability records. Centralized databases, ecosystem-specific platforms, fuzzing infrastructures, commercial aggregators, and issue trackers provide complementary metadata and artifacts, including affected versions, execution traces, advisories, dependency context, and patch references. Linking standardized identifiers to version-control histories and code revisions, these sources enable systematic study of vulnerability introduction, reporting, propagation, and remediation in open-source systems.

Centralized vulnerability sources. The National Vulnerability Database (NVD) [195], maintained by the National Institute of Standards and Technology (NIST), serves as a primary reference source for publicly disclosed vulnerability information by enriching CVE records issued through the CVE Program with structured metadata, including CWE classifications, CVSS base metrics, and CPE identifiers, thereby enabling standardized interoperability across security tools. However, analysis backlogs and enrichment delays can limit timeliness, and repository-level artifacts, such as fixing commits, precise code locations, or machine-readable version ranges aligned with the version-control history, are not consistently provided, necessitating correlation with external sources. For open-source ecosystems, the Open Source Vulnerability (OSV) Database [80] aggregates advisories from multiple upstream providers under a unified JSON schema designed around version-control semantics, explicitly modeling affected and fixed version ranges as well as introduced and fixed commit hashes across supported ecosystems. The GitHub Advisory Database

(GHSA) [75] similarly provides repository-centric advisories, linking vulnerabilities to source repositories, pull requests, issues, and discussions, and may include records without assigned CVE identifiers. Collectively, these platforms complement NVD by strengthening traceability between vulnerability disclosures and versioned source-code artifacts in open-source projects.

Complementary vulnerability sources. Fuzzing infrastructures such as OSS-Fuzz [202] complement vulnerability repositories by producing execution-based evidence of bugs through coverage-guided testing. Each detected issue typically includes a reproducer input and crash trace, from which a fix can be linked once a patch is committed; however, duplicate findings, non-security bugs, and nondeterministic failures require triage to ensure reproducibility and security relevance. Organization-specific repositories, including the Mozilla Foundation Security Advisories (MFSA) [167], publish product-scoped advisories that reference issue trackers such as Bugzilla, enabling alignment among advisories, bug reports, patches, and code diffs for causal analysis. Commercial aggregators, such as Snyk [213], Veracode [224], and Mend [156], extend public vulnerability data with proprietary analysis, including affected version ranges, fix versions, dependency context, and, in some cases, function-level granularity and reachability assessment. These services may disclose vulnerabilities prior to NVD enrichment and often include non-CVE entries and malicious packages. Their schemas and disclosure practices vary, requiring normalization when integrating multiple sources. The use of semantic versioning metadata and references to upstream repositories supports tracing vulnerabilities to specific commits. Issue trackers such as GitHub Issues, Jira, and Bugzilla [163] record discussions, patch reviews, and decision history associated with reported bugs. When correlated with advisories and commits, they provide evidence for reconstructing the technical processes involved in vulnerability introduction and remediation.

2.3 Related Work

This section reviews prior work spanning from vulnerability analysis to patch analysis. Section 2.3.1 examines taxonomies that structure vulnerability records into categories and descriptive attributes. Section 2.3.2 discusses research on vulnerability causality and representations of bug-error-failure chains. Section 2.3.3 reviews automated methods for assessing and enriching NVD records, highlighting both the utility and the limits of CVE descriptions. Section 2.3.4 then covers efforts to link vulnerability records to open-source repositories. Section 2.3.5 surveys work on identifying security patches and analyzing bug-fix patterns. Section 2.3.6 concludes by synthesizing these threads and positioning the gap addressed in this chapter.

2.3.1 Vulnerability Taxonomies in Empirical Analysis

Research on vulnerability analysis has been strongly shaped by taxonomies that organize vulnerabilities into structured categories and descriptive dimensions. These schemes classify vulnerability records using attributes such as *genesis*, *lifecycle phase*, *location*, *severity*, *platform*, *software type*, or *attack technique*, thereby imposing a descriptive framework on heterogeneous disclosures. As

summarized in Table 2.1, their main goals are to provide consistent naming schemes, support large-scale trend analysis, and enable predictive modeling. In this sense, they treat the *vulnerability entry*—typically derived from CVE/NVD data—as the primary unit of analysis and represent it as an entity characterized by descriptive traits. This makes taxonomies effective for aggregation and comparison across datasets, but it also means that they describe vulnerability properties rather than the underlying structure that links the enabling bug to its downstream manifestation.

Table 2.1: Summary Table of Reviewed Taxonomies.

Year	Taxonomy	Scheme	Attributes	Objective
1994	Landwehr *et al.* [126]	Multi-dimensional	genesis, time, location	establish taxonomy of security faults
2002	Frank Piessens [189]	Hierarchical	genesis, time	understand mistakes of software developers
2005	Weber *et al.* [230]	Hierarchical	genesis	aid designers of code analysis tools
2007	Bazaz & Arthur [12]	Hierarchical	resources, properties	strategies to evaluate software security
2017	Goseva & Tyo [85]	Multi-Dimensional	genesis, severity, location, time	build evidence-based vulnerability knowledge
2019	Garg *et al.* [69]	Multi-dimensional	genesis, severity, platform, technique	predictive modeling & preemptive mitigation
2020	Ezenwoye *et al.* [58]	Multi-dimensional	software type, weakness	reveal prevalence & persistence patterns
2022	Ezenwoye *et al.* [57]	Hierarchical	attack method & vector, technology	empirical taxonomy of web app. weaknesses

Multidimensional taxonomies. A prominent line of work organizes vulnerabilities along multiple dimensions in order to describe them from several complementary perspectives. Landwehr *et al.* [126] pioneered a multidimensional taxonomy capturing the genesis, time of introduction, and location of security faults, establishing a baseline for later frameworks. Goseva-Popstojanova and Tyo [85] linked software fault patterns (CWE-888 SFP View) to software lifecycle phases, producing data-driven vulnerability profiles that connect design and implementation stages. Garg *et al.* [69] extended this line by integrating severity, platform, and attack technique dimensions to support predictive modeling and mitigation strategies. Ezenwoye *et al.* [57] characterized vulnerabilities by software type, attack vector, and technology stack, identifying recurring weakness distributions across application domains in NVD data. More recently, Qian *et al.* [194] proposed a language-aware taxonomy comprising seven bug families, ranging from memory safety to side-channel vulnerabilities, relating programming language characteristics to detection paradigms. These approaches clarify recurring vulnerability traits and support pattern analysis, but they do not represent the internal structure through which a vulnerability is enabled and manifests.

Hierarchical taxonomies. Complementing multidimensional schemes, hierarchical taxonomies structure vulnerabilities into layered abstractions that group specific variants into high-level categories. Building on Landwehr's framework [126], Piessens [189] proposed a hierarchical taxonomy emphasizing vulnerability causes to guide vulnerability identification during security review. Weber *et al.* [230] adapted Landwehr's structure for static code analysis, pruning categories not directly observable at the source-code level to better align with tool design. Bazaz and Arthur [12] introduced a verification-oriented taxonomy organized around computational resources (e.g., memory, I/O, cryptographic elements) and the assumptions or constraints that can be violated. At a larger empirical scale, Ezenwoye *et al.* [58] classified NVD entries into seven software types, demonstrating prevalence patterns of weakness classes across domains. Hierarchical models can reveal some structure around vulnerabilities, but they do not capture the vulnerability structure at the code-level or how it propagates.

2.3.2 Vulnerability Causality and Propagation Chains

Understanding how software vulnerabilities emerge and manifest requires representations that connect the underlying bug to the resulting error and failure. Prior research has examined these relations at different levels of granularity, including studies of root causes, consequences, impacts, and, in some cases, using BF for expressing vulnerability chains. This line of work is relevant because it moves beyond classifying vulnerability traits toward their internal structure. However, relatively few studies reconstruct vulnerability chains, and fewer do so in ways that support characterization of the underlying bug across the software context in which it occurs.

Causal analyses of vulnerabilities. Jia et al. [109] analyzed 202 TensorFlow bugs and linked root causes to consequences using a custom taxonomy, observing that root causes such as type confusion or dimension mismatch more consistently predict consequences (e.g., functional errors, crashes) than the reverse. Shao et al. [204] examined 1,119 bugs across 17 federated learning frameworks and identified 12 root causes, 12 impacts, and 18 fix patterns, finding that logic and assignment mistakes were the dominant causes and manifested in diverse failure modes, including privacy leaks and performance degradation. From a security perspective, Goseva-Popstojanova and Tyo [85] studied 664 security issues in NASA mission systems using the CWE Software Fault Pattern taxonomy and found that most vulnerabilities originate during implementation and cluster within a limited set of subsystems, with five CWE categories accounting for most cases. These studies show that recurring relations between causes and manifestations can be identified empirically, but they stop short of representing vulnerabilities as propagation chains.

Modeling vulnerability chains. A smaller body of work models vulnerability chains more explicitly. Harzevili et al. [93] manually analyzed vulnerability-fixing commits in ML libraries and mapped root causes to observed outcomes, but their terminology blurs the distinction between root cause and manifestation. Bojanova and Galhardo [20] reconstructed HEARTBLEED (CVE-2014-0160) using the BF, showing how two distinct causal chains can result in Information Exposure,

but their analysis is limited to a single case and still models both program code and program state within the same unit of analysis. At the automated level, James et al. [107] used static and dynamic analysis to construct backward taint graphs and generate BF explanations for the full bug–crash chain, correctly diagnosing over 94% of tested vulnerabilities, but their approach is restricted to crashing inputs in C/C++ programs and therefore covers only a narrow subset of memory-related vulnerabilities. These studies show the value of modeling vulnerability chains, but they remain limited in scope and do not characterize the underlying bug across the software context.

2.3.3 Automated Analysis and Enrichment of Vulnerability Records

Empirical vulnerability analysis depends heavily on vulnerability databases such as NVD, yet these records are often incomplete, inconsistent, and delayed. This has motivated extensive research on automating vulnerability analysis by extracting structured concepts from CVE/NVD descriptions and predicting missing metadata such as CWE and CVSS attributes. Such methods improve the structure, consistency, and quality of vulnerability records, making them more usable for large-scale analysis. However, because they operate primarily on reported descriptions and metadata, they recover what is documented about the vulnerability—most often its *consequences*, *impacts*, and related attributes—rather than the *root cause* when it is not explicitly described.

Reliability of vulnerability reports. The integrity of vulnerability databases directly constrains the validity of empirical security research [128, 41]. Large-scale analyses of CVE and NVD records reveal substantial omissions, inconsistencies, and reporting delays. Guo et al. [88] report that over 50% of entries lack *vulnerability type*, 85% omit *root-cause*, and nearly 40% do not specify the *attack vector*. Anwar et al. [6] identify reporting delays exceeding one week in 28% of cases and *severity* inconsistencies across CVSS versions in 40%, alongside missing or inconsistent vendor and product labels. Cross-source comparisons further expose reliability concerns: fewer than 60% of NVD entries align with corresponding CVE summaries and external references [52], and roughly 25% of examined Chrome-related entries were found erroneous [172]. *Severity* scoring has also been scrutinized, with discrepancies between expert assessment and CVSS metrics reported by Holm and Afridi [99] and Johnson [112], and downstream performance degradation of automated *severity* models reaching 71% under inconsistent annotations [42]. These findings show that vulnerability records cannot be treated as uniformly reliable inputs, motivating automated methods to normalize, complete, and reconcile their contents.

Extraction of vulnerability concepts. CVE descriptions follow standardized reporting templates that encourage inclusion of *vulnerability type*, *root cause*, *impact*, *attacker type*, and *attack vector* [161]. Although concise (averaging roughly 37.5 words) [133, 90], they encode dense, domain-specific information in free-text form. Their hybrid linguistic–technical nature, combined with increasing disclosure volume [25], makes manual structuring error-prone and susceptible

to ambiguity and variability [7, 248]. To address this, a substantial body of work has developed automated extraction techniques that transform unstructured CVE/NVD text into machine-readable attributes. Early approaches applied Named Entity Recognition (NER) to identify *products*, *versions*, and *weakness* categories, including few-shot fine-tuning of pre-trained language models [242] and hybrid rule-based pipelines [118]. Supervised NER models trained on annotated corpora achieved F1-scores exceeding 85% for product entities and 73% for *weakness* terms [125]. Recent efforts attempt reducing labeling overhead through unsupervised clustering of CVE descriptions [248], traditional machine-learning classifiers for *attack vector* and *impact* prediction [177], and question-answering with pre-trained language models to extract vulnerability concepts [231]. These methods make reported vulnerability concepts more accessible for analysis, but they remain limited to information expressed in the description and therefore do not recover the *root cause* when it is omitted.

Prediction of vulnerability metadata. To infer missing or delayed metadata in CVE/NVD records, such as CWE identifiers or CVSS metrics, a substantial body of work has proposed predictive approaches [65, 79]. Neural classifiers have been used to predict key vulnerability concepts with F1-scores between 70.6% and 94.6% [88]. Random Forest and Transformer-based models have achieved over 90% F1-scores in CWE prediction [7, 47]. Neural architectures have also demonstrated strong performance in estimating CVSS severity and related metrics such as *attack vector* and *user interaction* [90, 125]. Predictive modeling has also been extended to *product* identification [229] and *exploitability* assessment [178]. Prompt-based learning with pre-trained language models has outperformed fine-tuned baselines for both *severity* and *exploitability* prediction [133]. Predictive models can fill important gaps in vulnerability metadata, but they still infer record-level labels rather than reconstructing the code-level bug that enabled the vulnerability.

2.3.4 Linking Vulnerability Records to Open-Source Repositories

Recovering the underlying bug of a vulnerability requires access to the source code artifacts of the affected product. However, databases such as NVD identify vulnerable products primarily through CPE metadata rather than direct links to the open-source repository versions where the vulnerability is introduced and patched. Although CPE provides a naming scheme for affected products, it does not indicate whether the relevant product is open-source or which repository contains the enabling bug when multiple products are affected. Research on ecosystem mining and vulnerability linking addresses this gap by connecting vulnerability records to packages and repositories across software ecosystems. These efforts are essential for recovering the code artifacts needed for reconstruction, but mapping vulnerability records to versioned repositories remains incomplete and depends on reconciling inconsistent naming and metadata across databases.

Mining vulnerabilities in open-source ecosystems. Empirical research has increasingly mined open-source ecosystems to analyze vulnerability propagation, dependency exposure, and disclosure dynamics. Pashchenko et al. [184] reported that roughly 20% of vulnerable Java dependencies

were not actually deployed and that many issues could be mitigated through straightforward version updates. Ecosystem-specific studies further examined disclosure timelines and dependency risk: Zerouali et al. [253] observed rising vulnerability frequency alongside faster disclosure in npm, Alfadel et al. [2] found an average time-to-discovery exceeding three years for Python packages, and subsequent analysis of 6,546 Node.js applications showed that nearly 68% depended on vulnerable components [3]. More recent work applies machine learning to identify affected libraries and ecosystems, including VULLIB, which achieves an average F1-score of 0.657 [32], and a learning-to-rank framework that achieves a mean average precision@1 of at least 85.5% [237]. Large-scale advisory analysis further indicates rapid growth in reported OSS vulnerabilities, with a 98% annual increase across GitHub and Snyk datasets [1]. These studies improve understanding of vulnerability disclosure within open-source ecosystems, but they do not, by themselves, resolve the linkage between vulnerability records and the repository artifacts needed for reconstruction.

Linking vulnerabilities to open-source repositories. Accurate linkage between vulnerability records and their open-source repositories is essential for recovering the code in which the bug resides. CVE entries reference affected products using the CPE dictionary, yet this metadata is often inconsistent or incomplete. Sanguino and Uetz [198] identified missing or incorrect CPE references, naming ambiguities, typographical errors, and version mismatches that impede automated mapping to software repositories. Dam and Neumaier [45] evaluated three name-matching approaches to connect CVEs with LIBRARIES.IO packages, finding that although matching improved precision, overall coverage remained constrained by absent repository identifiers and ambiguous product naming. Industry initiatives aim to mitigate these limitations by standardizing package identifiers. The Package URL (PURL) specification encodes *ecosystem*, *package*, and *repository* information in a uniform format, and projects such as SCANOSS purl2cpe [201] provide mappings between PURLs and CPEs to facilitate automation. OSV also uses PURLs to specify affected and fixed versions across supported ecosystems [80], while GHSA links vulnerabilities to repositories, pull requests, issues, and discussions, and may cover cases that lack assigned CVE identifiers [75]. These efforts bring vulnerability analysis closer to code bases, but automated linkage remains constrained by incomplete and inconsistent product metadata across vulnerability reporting and software ecosystem databases.

2.3.5 Security Patch Identification and Analysis for Bug Localization

Security patches provide the most direct artifact for locating the code changes that addressed the bug of a disclosed vulnerability. Prior work identifies vulnerability-fixing commits, constructs curated patch datasets, and characterizes recurring repair patterns to study remediation practices and support downstream tool evaluation. This line of work is important because the patch reveals what was wrong in the code and therefore offers strong evidence for localizing the underlying bug. However, patches are not a direct representation of the bug itself: they may also include refactoring, tests, or hardening changes, and prior work largely analyzes them in terms of how vulnerabilities are fixed rather than how the enabling bug should be recovered.

Dataset construction and identification of security patches. Representative dataset construction efforts recover security fixes primarily by reusing references already embedded in vulnerability metadata. Ponta et al. [193] combine NVD disclosures with project-specific sources to curate fix links, while Fan et al. [61] build Big-Vul by tracing CVE records to repository and release references, yielding 3,754 vulnerabilities and 4,432 relevant commits. Bhandari et al. [13] automate this idea in CVEFIXES, collecting 5,365 CVEs and 5,495 vulnerability-fixing commits. Ye et al. [245] further scale metadata-based recovery by filtering 217,403 NVD entries for patch references, but ultimately extract fixes for only 8,487 vulnerabilities. These studies show that patch datasets can be built effectively from existing metadata, yet such pipelines still recover only a limited subset of disclosed vulnerabilities. For CVE records that lack explicit links to fixes, later work attempts to recover them from repository histories. Li et al. [132] proposed PatchFinder, a two-stage retrieval and reranking approach that reports Recall@1[1] of 79.23% on 4,789 CVEs. Nguyen et al. [171] extended this line of work by combining NVD references, external security databases, and GitHub mining, compiling 37,441 vulnerability-fixing commits across 26,710 NVD records; on their evaluation dataset, PatchFinder's learning-based approach achieved Recall@1 of 22.12%, while Prospector's heuristic approach reached 75.61%, suggesting that commits are more reliably recovered when they can be associated through vulnerability, issue, or advisory IDs, or information in descriptions and affected code. These results indicate that fix links can often be recovered beyond explicit metadata links, but they still depend substantially on direct linking evidence.

Patch analysis for bug fixing patterns. A related line of work analyzes patches themselves to characterize how bugs are corrected in code. Early studies of general bug fixing found that patches are often small and localized: Martinez et al. [153] identified recurring change patterns across 23,597 Java revisions, Zhao et al. [257] observed that most faulty functions are corrected through one or two dominant change types, and Sobreira et al. [214] found that 92% of patches in DEFECTS4J modify a single file. The CODEFLAWS dataset [210] similarly classifies over 3,900 defects based on syntactic differences between buggy and fixed programs, while Ni et al. [173] relate fix structures to bug causes through tree-based neural models. Security-related studies report similar locality but emphasize remediation tactics such as validation and sanitization. Analysis of 3,663 patched vulnerabilities across 1,096 GitHub projects showed that most fixes are localized and often add validation checks or sanitization logic [102], while Harzevili et al. [93] identified 22 recurring fixing patterns in machine-learning libraries and Zhang et al. [254] developed a hierarchical taxonomy of 44 remediation tactics from 21,187 GitHub issues. Together, these studies show that vulnerability fixes exhibit recurring and often localized repair patterns, but they characterize the corrective changes rather than the underlying bug that made those changes necessary.

[1] We report Recall@1 here to emphasize exact identification of the fixing commit.

2.3.6 Summary and Research Gap

Vulnerability research treats publicly disclosed records as the primary unit of analysis, characterizing cases through traits that support consistent aggregation and trend analysis. However, a smaller body of work shows that vulnerabilities are better understood as bug-error-failure chains, and that such representations can reveal relations between root causes and manifestations. Automated vulnerability analysis and enrichment extract concepts and predict missing metadata from CVE descriptions to support empirical analysis, but the recovered information largely reflects consequences and impacts, because those are the aspects most often described in the text. The disclosed record, therefore, captures only the manifestation side of the vulnerability chain, leaving the underlying bug unidentified. Characterizing that bug requires shifting the unit of analysis from manifestation to cause, which in turn requires access to the source-code artifacts of the affected product and motivates work that links vulnerability records to open-source repositories. Within that artifact space, security patches provide the clearest evidence for bug recovery, since they reveal the code changes that resolved the vulnerability and thus offer the strongest basis for inferring what was wrong in the code.

Prior work still does not provide a reliable path from disclosed manifestation to the underlying bug, reflecting a broader tendency to represent vulnerabilities in terms of observable outcomes rather than the root cause. Most studies characterize vulnerabilities through labels that capture reported traits, while only a few studies model them as bug-error-failure chains, and even fewer characterize the underlying bug. This reflects a broader limitation of vulnerability descriptions, which typically report the manifestation while omitting the root cause. Recovering that root cause requires access to the repository of the affected software, yet existing linkage remains incomplete because product metadata is often noisy and imprecise. Even when fixing patches can be recovered, they are not a direct representation of the bug itself, as they may also include refactoring, tests, or hardening changes, and prior work largely analyzes them in terms of bug-fixing patterns.

This gap motivates the research question of this chapter (RQ1): **What are the profiles of software security bugs and their manifestations?** Addressing this question requires shifting the unit of analysis from the manifestation to the root cause, so that the bug can be characterized separately from the errors and failures through which the vulnerability becomes visible. The chapter responds by reconstructing vulnerabilities into a representation that captures root cause, software context, and manifestation, enabling recurring security bugs to be characterized across the settings in which they arise and related to the downstream behaviors they produce.

2.4 Methodology

The methodology, summarized in Figure 2.3, begins with a curation and validation stage (Section 2.4.1) that verifies CWE, CPE, and CVSSv3 metadata, ensuring that only consistent and sufficiently enriched CVE records are used for analysis. Because reconstructing comparable vulnerability chains is not feasible from arbitrary records, the methodology retains only cases that

support reliable reconstruction, resulting in a relatively small but usable corpus. Building on the validated metadata, the process identifies vulnerabilities whose exploitability is confined to the vulnerable component (Section 2.4.2), retaining cases whose activation depends on a few conditions beyond reaching the vulnerable code. The subsequent stage further delimits the corpus to high-severity vulnerabilities whose effects remain within the affected component and for which the code evidence needed for reconstruction is available through open-source applications with traceable repositories and accessible patches (Section 2.4.3). The methodology then reconstructs *bug* → *error* → *failure* chains by extracting manifestation details from vulnerability descriptions and inferring the bug from the corresponding patch (Section 2.4.4). Finally, it applies a labeling schema that represents each case through *root cause*, *software context*, and *manifestation* (Section 2.4.5). Collectively, these stages transform heterogeneous vulnerability records into structured and labeled case representations that support bug profiling.

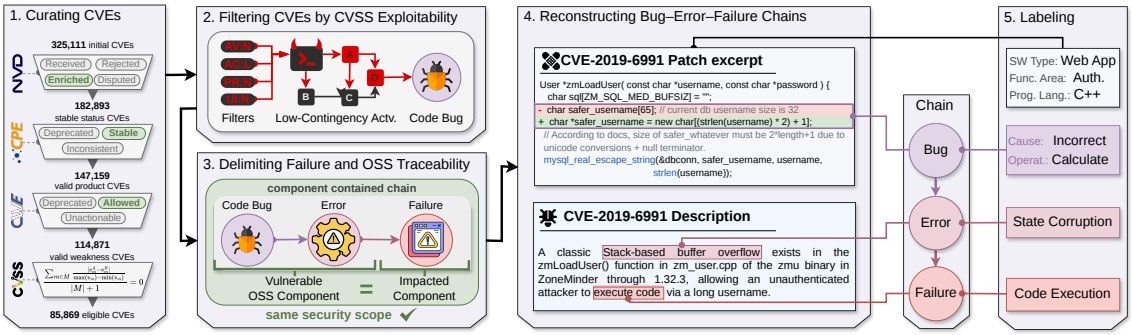


Figure 2.3: Overview of the methodology: (1) CVE curation, (2) exploitability filtering, (3) failure delimitation and OSS traceability, (4) bug-error-failure chain reconstruction, and (5) labeling the resulting cases by root cause, software context, error, and failure.

2.4.1 Curating and Validating CVE Metadata for Eligibility

An initial curation and validation stage retains only CVE records with sufficiently enriched and internally consistent metadata for further consideration. It serves as a quality-control stage over the National Vulnerability Database (NVD) corpus by validating four metadata elements: *Status*, *CPE*, *CWE*, and *CVSS*. Entries are first screened by record status to exclude CVEs with provisional, rejected, deferred, disputed, or otherwise unstable metadata, retaining only records that reflect a completed analysis state (Appendix A.1.1). The next step in this stage requires usable product metadata so that the affected software can be traced to a concrete repository and later linked to *software type*; to this end, CPEs are normalized against a fixed CPE Dictionary snapshot, and deprecated or internally inconsistent product references are removed (Appendix A.1.2).

Next, weakness identifiers are validated by requiring at least one active CWE mapping that is not discouraged by root cause guidance [162], so that the weakness mapping remains interpretable and comparable across cases (Appendix A.1.3). Finally, CVSSv3 exploitability and impact metrics are checked for sufficient consistency to support downstream filtering. When multiple CVSS vectors are present across sources, cross-source divergence is quantified using a normalized

disagreement score over base metrics, and cases with inconsistency greater than 0 are excluded (Appendix A.1.4). The four validations are applied sequentially, and a CVE is retained only if it passes all stages. Figure 2.4 summarizes the resulting attrition: of 325,111 CVEs in the initial NVD set, 182,893 pass the status validation, 147,159 have usable CPE data, 114,871 satisfy the CWE eligibility rules, and 85,869 remain after excluding CVEs with inconsistent CVSSv3 metrics.

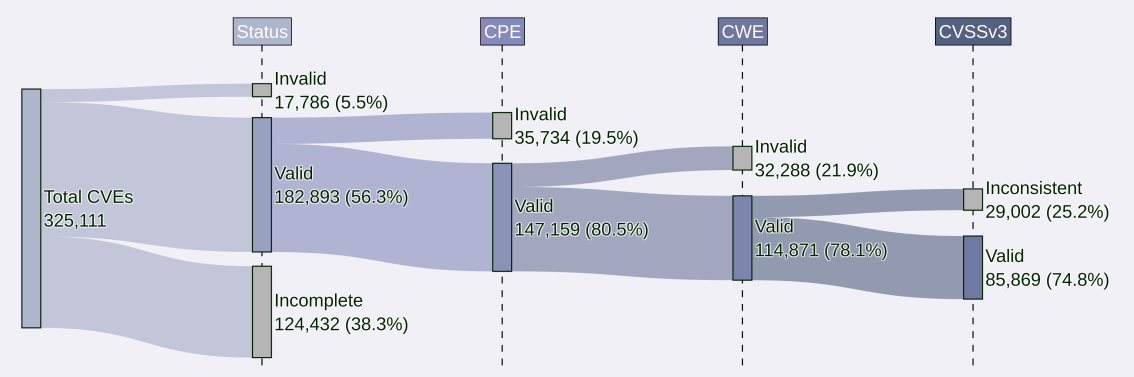


Figure 2.4: Sankey flow summarizing triage of CVEs by *Status*, *CPE*, *CWE*, and *CVSSv3*.

2.4.2 Filtering CVEs by Exploitability for Code-Level Bugs

To support later reconstruction of vulnerabilities whose root cause is at the code level, the curated CVE set is further narrowed to vulnerabilities whose activation depends on a few conditions beyond the code of the vulnerable component. The analysis targets cases where activation is primarily driven by externally supplied input, does not require user interaction or attacker privileges, and does not depend on unusual environmental setup, so that later reconstruction can focus on the code of the vulnerable component rather than on external preconditions.

To operationalize this selection, fixed inclusion constraints are applied over the CVSSv3 exploitability fields to reduce variation in the conditions required to activate the vulnerable code, favoring cases in which the vulnerable path can be reached directly through the code of the vulnerable component rather than through additional operational contingencies. Specifically, only CVEs scored as *Attack Vector: Network*, *User Interaction: None*, *Privileges Required: None*, and *Attack Complexity: Low* are retained (Appendix A.1.5). Figure 2.5 summarizes the resulting attrition: 65,434 CVEs remain after constraining *Attack Vector* to Network, 38,304 after requiring no *User Interaction*, 25,259 after requiring no *Privileges Required*, and 23,449 after restricting *Attack Complexity* to Low.

2.4.3 Delimiting Failure Boundary and OSS Traceability

This stage starts by constraining the failure boundary and impact regime of the corpus. It requires *Scope: Unchanged* in CVSSv3 so that the impacted component remains within the same security

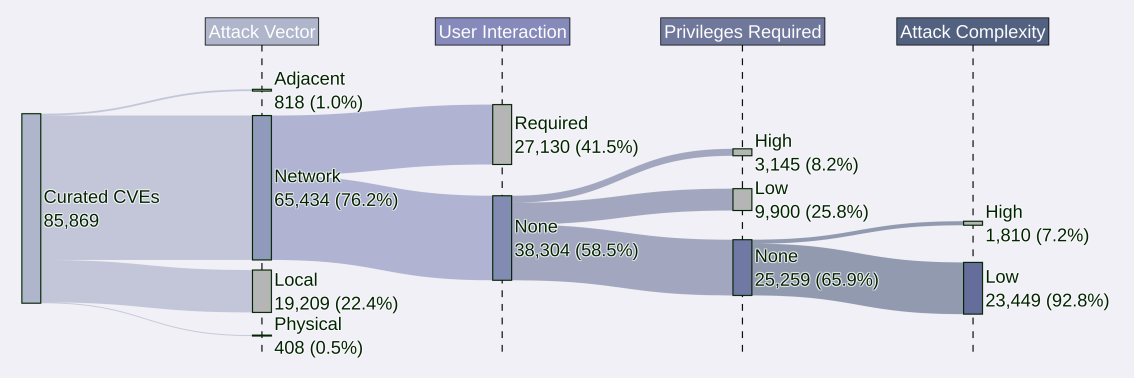


Figure 2.5: Sankey flow summarizing triage of CVEs by *exploitability* CVSSv3 metrics.

scope as the vulnerable component, avoiding cross-scope effects that would blur the causal boundary of the *bug* → *error* → *failure* chain. It also retains only CVEs with *Base Severity: Critical* to normalize the corpus to a common impact regime, so that later comparisons are driven by bug characteristics rather than broad variation in severity (Appendix A.1.5).

Next, this stage restricts the dataset to open-source application components for which the underlying bug can be traced through public code artifacts. Using CPE metadata, it retains only CVEs affecting application components and excludes cases involving operating systems or hardware, since mixing component types can introduce compound cases in which the vulnerability arises from interacting components, complicating attribution. To establish traceability to open-source repositories, affected products are resolved using complementary linkage signals, including repository references in product metadata and curated mappings between CPE identifiers and package URLs. Only cases whose repositories can be resolved and accessed for patch-based reconstruction are retained; the specific linkage sources, resolution logic, and repository validation checks are detailed in Appendix A.1.6. Figure 2.6 summarizes the resulting attrition: of the 23,449 confined CVEs, 23,021 satisfy the unchanged-scope constraint, 11,371 are classified as *Critical*, 9,952 affect application components, and 4,324 can be resolved to accessible open-source repositories, leaving 5,628 unresolved at the repository-linkage stage.

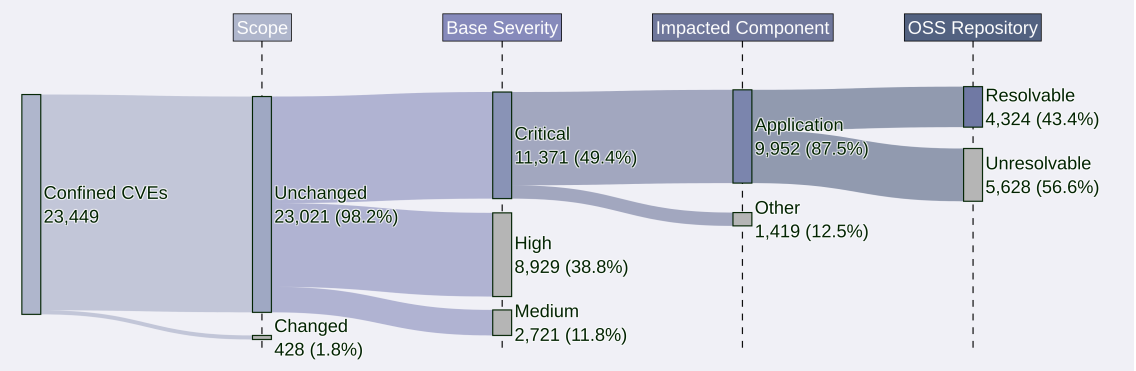


Figure 2.6: Sankey flow summarizing triage of CVEs by *severity* and *OSS traceability*.

2.4.4 Reconstructing Bug–Error–Failure Chains from Artifacts

This stage reconstructs the *root cause* and *manifestation* for each retained CVE by combining manifestation details from the vulnerability description with bug evidence from the corresponding patch. The CVE description is treated as the primary source of manifestation cues because public reports more often describe the consequence and impact of a vulnerability than its root cause [162]. Requiring explicit manifestation details in the description also functions as a quality screen: CVEs that do not provide enough information about the observed weakness and its impact cannot reliably support reconstruction of the vulnerability chain.

The first step in the stage screens and parses CVE descriptions to extract a minimal set of structured entities, since descriptions often follow phrasing patterns and reuse recurring vulnerability language. It combines basic text-quality checks with pattern-based rules to identify *weakness* and *impact* signals, which are later mapped to *error* and *failure*, and to discard descriptions that are malformed or underspecified. The concrete description screening rules and extraction patterns are reported in Appendix A.1.7. Figure 2.7 illustrates this step for CVE-2019-6991, drawn from the subset of descriptions where both *weakness* and *impact* are explicitly stated. The description provides the two required manifestation elements—*stack-based buffer overflow* and *execute code*—together with contextual entities such as the affected product (*ZoneMinder*), forming the basis for reconstructing the manifestation side of the bug–error–failure chain.

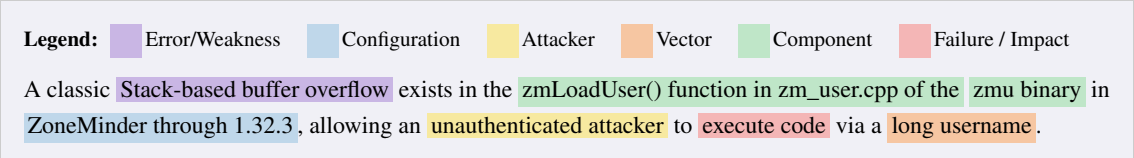


Figure 2.7: CVE description with extracted entities used for reconstruction (CVE-2019-6991).

After this extraction step, the corresponding patch is located and validated by following NVD references into the previously resolved OSS repositories, retaining only cases where the patch can be reliably accessed and associated with the affected project; the patch evidence is then used to localize and identify the bug. The extraction patterns and patch validation checks are reported in Appendix A.1.8. For CVE-2019-6991, automated resolution via the referenced issue (2478) does not yield a commit through the Application Programming Interface (API), requiring manual identification of the fix (commit 5b39eaa). Figure 2.8 shows the corresponding diff, which modifies the same file (`src/zm_user.cpp`) and function (`zmLoadUser()`) identified in the description. The change replaces a fixed-size stack buffer with an allocation derived from the input length, providing the evidence used to infer the underlying root cause. In cases with multiple candidate files, diff-to-text summaries and commit metadata are used as heuristic signals to prioritize files based on similarity to the original commit message and fix-type cues, providing an entry point for subsequent manual validation; in this example, the single modified file is selected directly.

Figure 2.9 shows the resulting attrition: of the 4,324 critical OSS CVEs, 1,819 have description text that is compliant for downstream processing, while 1,993 are multi-sentence and 512 are

```
...
User *zmLoadUser( const char *username, const char *password ) {
  char sql[ZM_SQL_MED_BUFSIZ] = "";
-  char safer_username[65]; // current db username size is 32
+  char *safer_username = new char[(strlen(username) * 2) + 1];

  // According to docs, size of safer_whatever must be 2*length+1 due to
      unicode conversions + null terminator.
  mysql_real_escape_string(&dbconn, safer_username, username, strlen(
     username ) );
...
```

Figure 2.8: Patch excerpt for CVE-2019-6991; context lines collapsed.

filtered for unacceptable grammar. From that subset, 924 cases match the phrasing patterns and 895 do not. Among the matched cases, 451 are retained after the extraction step and 473 are excluded. Finally, only 66 of the 451 retained cases have a resolvable patch, while 385 do not.[2]

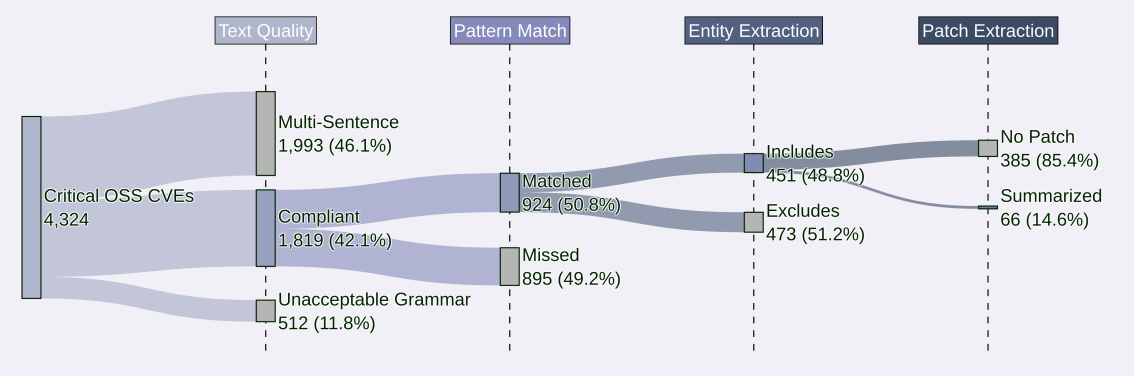


Figure 2.9: Sankey flow summarizing triage of CVEs by *description* and *patch* extraction.

Although this stage yields a small final subset, the objective is not broad statistical coverage of reported vulnerabilities but reliable reconstruction; the observed attrition also exposes which description and artifact gaps prevent many CVEs from supporting this type of analysis.

2.4.5 Vulnerability Representation and Labeling Schema

Each reconstructed vulnerability is represented through three complementary elements: *root cause*, *software context*, and *manifestation*. In this schema, *root cause* and *software context* form the basis for later bug profiling, enabling recurring bug patterns to be analyzed across software areas, while *manifestation* normalizes the observed consequences in the CVE description into a common analytical form, reducing variation due to disclosure style. This structure separates cause from consequence and implementation setting, allowing cross-case comparisons to reflect differences

[2]No diff files retrieved for CVE-2018-11652, CVE-2018-12578, CVE-2020-19138, CVE-2022-41642, CVE-2022-42496, CVE-2023-39001, CVE-2023-39008.

in vulnerabilities rather than artifacts of reporting. The classifications were refined iteratively through consistency checks before being applied across the corpus, and the annotation was manual [3] (disambiguation notes and procedural details are provided in Appendix A.2).

Each case is then annotated along the *manifestation* dimension using information derived from the CVE record. An *error* label is assigned by first checking that the weakness stated in the description aligns with the primary CWE recorded in NVD metadata, and then mapping that CWE into the error classification (Appendix A.2.2), which is grounded in Bazaz and Arthur's process/object model of computation [12] but reorients that abstraction toward runtime state deviation. A *failure* label is assigned by mapping the impact phrasing in the description into the failure classification (Appendix A.2.3), grounded in the common impact space shared by multiple taxonomies and frameworks [160, 159, 158, 170, 16], and by resolving multi-consequence reports to a single dominant outcome. Figure 2.10 illustrates this step for CVE-2019-6991. The description provides both required manifestation elements: the weakness (*stack-based buffer overflow*) and the impact (*execute code*). These are mapped to *State Corruption* and *Code Execution*, respectively, with each label tied directly to the corresponding evidence in the text.

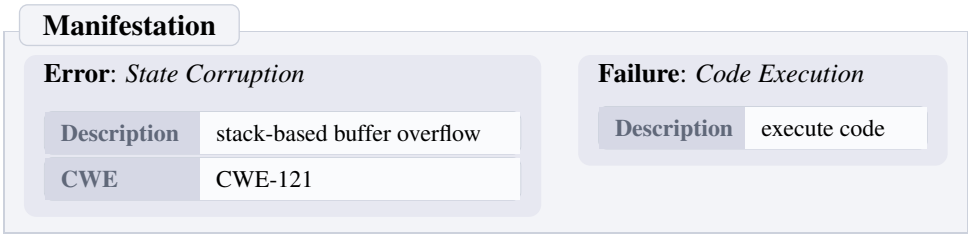


Figure 2.10: Manifestation labeling for CVE-2019-6991 based on the CVE description.

To capture the *root cause*, the patch diff is inspected and both a *cause* and an *operation* label are assigned, such as *Missing Validate* or *Incorrect Calculate*. The root cause classification is derived from MITRE's CWE glossary [39] for *cause* labels—drawing on the distinction between behaviors not attempted (**Missing**) and those attempted but flawed (**Incorrect**)—and from NIST's Bugs Framework [16] for *operation* labels. When the relevant BF operation is not explicit in the vulnerable code, CWE is used to align the fix with the closest operation; the complete labeling scheme, patch-selection procedure, and disambiguation rules are provided in Appendix A.2.1. Figure 2.11 illustrates this step for CVE-2019-6991. The vulnerable code allocates a fixed-size buffer (`char safer_username[65]`) based on an assumption about input length (commented as "current db username size is 32"), while the comment for `mysql_real_escape_string()` specifies that the buffer must satisfy $2 \times \text{length} + 1$. The constant 65 is consistent with that rule for a 32-character input, but is applied as a fixed bound rather than computed from the actual input length. Together with the patch—which replaces the allocation with one derived from `strlen(input)`—this indicates that the required size computation is understood but not correctly realized, yielding an

[3] Machine-learning classification was explored, but the lack of labeled data and the dataset size made automation costly relative to its expected benefits.

Incorrect classification for the CALCULATE operation [4]. Since this computation is implicit in the vulnerable version, the alignment with BF is supported through CWE-131, which captures incorrect buffer size calculation.

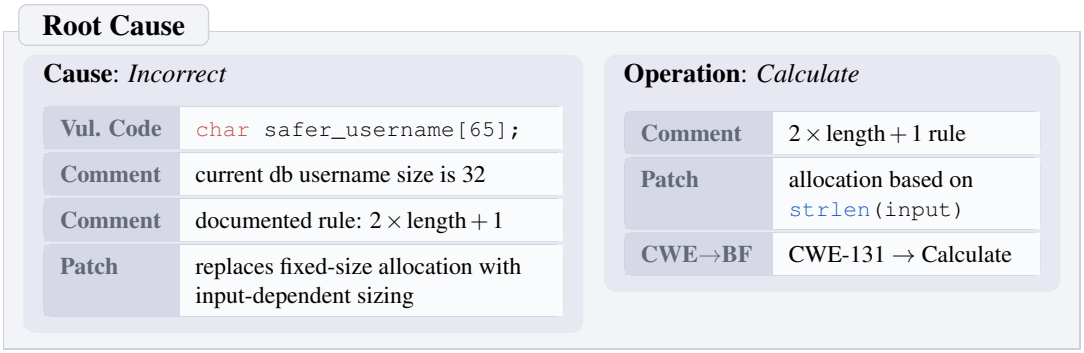


Figure 2.11: Root-cause labeling for CVE-2019-6991 based on the patch analysis.

Finally, each case is labeled with *software context* derived from implementation artifacts to capture the setting surrounding the bug. A *software type* label is assigned using the software type classification (Appendix A.2.4), adapted and extended from Ezenwoye et al.'s classification [58] to cover the main ways software is deployed, run, or used. A *functional area* label is assigned using the functional-area classification (Appendix A.2.5), adapted from CWE's *Functional Area* vocabulary by excluding out-of-scope values, refining some label names, and assigning labels from the defective code region in the patch rather than from the CWE. Figure 2.12 illustrates this step for CVE-2019-6991. The project is classified as a *Web Application*, as the documentation identifies the `/zm` web console as the primary interface through which users access and operate the system. The vulnerable code is located in `zmLoadUser()` within `src/zm_user.cpp`, where username and password are used to authenticate users, placing the bug in the *Authentication* functional area. The programming language is recorded as *C++*, based on the modified vulnerable file.

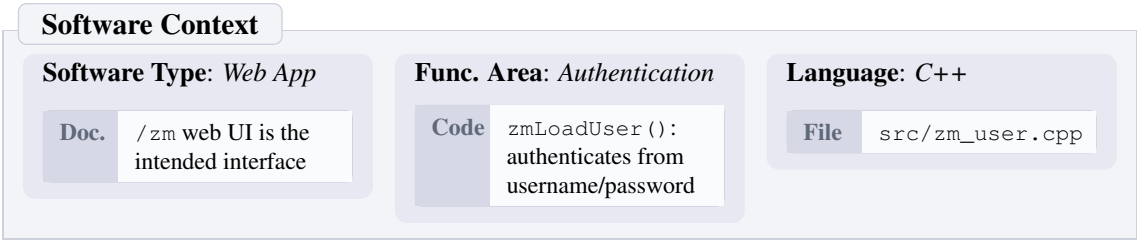


Figure 2.12: Software-context labeling for CVE-2019-6991 based on the repository artifacts.

[4] Under BF, this case could also be interpreted as MAL/Allocate, since that class covers bugs in reserving memory, including assigning an object an incorrect size before it is used. However, it is not used as the primary label because the corresponding CWE space (uncontrolled resource consumption) does not align cleanly here (undersized-buffer).

2.5 Results

Using the curated and labeled corpus produced by the workflow in Section 2.4, this section reports the observed profiles of software security bugs. The analysis focuses on the subset of CVEs for which the *bug* → *error* → *failure* chain was successfully reconstructed from artifacts. Section 2.5.1 summarizes the resulting dataset and labeling coverage. Section 2.5.2 addresses RQ1 by identifying the recurring bug patterns across software context areas, while distinguishing the occurrences by *Missing* and *Incorrect* causes. Finally, Section 2.5.3 analyzes how the identified bug profiles propagate through error and failure outcomes.

2.5.1 Dataset Overview

A total of 60 candidate CVEs were analyzed for reconstruction from artifacts, of which 55 were retained after case-level eligibility checks. Five cases were excluded during manual reconstruction: two were removed as duplicates (CVE-2020-20914, duplicated by CVE-2020-20915; and CVE-2023-36258, duplicated by CVE-2023-36095), two were excluded because the referenced patch contained extensive unrelated changes that prevented reliable bug localization and root cause attribution (CVE-2023-26861 and CVE-2020-36052, the latter closing 17 issues and modifying 3,990 lines), and one was excluded because it reflected a design issue rather than a coding bug suitable for the analysis (CVE-2024-23687). All retained cases admit complete reconstruction, including an identified bug, labeled *error* and *failure*, and associated *software context*.

The retained corpus spans 7 software types—*Web App*, *Library*, *Client*, *Server*, *Extension*, *Utility*, and *Framework*—and 7 programming languages: *PHP*, *C*, *C++*, *Java*, *JavaScript*, *Python*, and *Go*. To characterize the setting of the bug, 10 functional areas were labeled: *File Processing*, *String Processing*, *Number Processing*, *Program Invocation*, *Logging*, *Security Enforcement*, *Authentication*, *Networking*, *Error Handling*, and *Resource Management*. The improper code construct underlying the bug is represented through 8 operation categories—VALIDATE, VERIFY, SANITIZE, CALL, EVALUATE, DEREFERENCE, CALCULATE, and DECLARE. On the manifestation side, the cases cover 3 error categories—*Semantic Corruption*, *State Corruption*, and *Authority Violation*—grounded in 17 unique primary CWE identifiers derived from the reported weakness, and 4 failure categories: *Code Execution*, *Denial of Service*, *Tampering*, and *Information Exposure*, grounded in the reported impact.[5]

Corpus size in context. This size of the corpus reflects the manual effort required to inspect patches and related artifacts for each case. Wei et al. [232] analyzed **318 vulnerabilities with fixing links** within a dataset of **1,076 CVE reports**, with patch-level inspection only for that subset, while the remaining cases were classified primarily from CVE/NVD descriptions and metadata. Harzevili et al. [93] manually reviewed **683 vulnerability-fixing commits**, examining commit messages, code changes, issues, and pull requests to identify patterns, including **36 CVEs**. James et al. [107] curated **33 vulnerabilities mapped to CVEs**, manually inspecting bug reports, fixing

[5]The failure classes in this thesis (see Appendix A.2.3) overlap considerably with the failure classes of BF [16].

commits, and reproducing the failures to determine their causes and locations. Together these studies show that as analysis progresses from description classification to commit inspection and then to reproduction of failures, the number of vulnerabilities that can be examined decreases.

2.5.2 Profiles of Software Security Bugs

To address the first part of RQ1, this subsection characterizes profiles of software security bugs in the corpus through three analytical questions. AQ1 examines variation by *software type*, capturing the architectural setting in which bugs arise; AQ2 by *functional area*, capturing where the bug is located within the software; and AQ3 by *programming language*, capturing implementation-level constraints and constructs. Across these analyses, operation categories provide a common representation, and distributions are reported separately for the **Missing** and **Incorrect** causes. The resulting security bug profiles are represented as tuples of *cause*, *operation*, *software type*, *functional area*, and *programming language*.

AQ1. How do security bugs vary across software types?

> ***Intent:*** *Examine how software shapes concentration and diversity of bugs by root cause.*

Figure 2.13a summarizes the distribution of security bugs by root cause across software types in the retained corpus and shows a more concentrated pattern for **Missing** bugs. Three operations account for all cases—VERIFY (47.5%, 19/40), VALIDATE (40.0%, 16/40), and SANITIZE (12.5%, 5/40). *Web applications* account for most of these cases (27/40, 67.5%) across all three operations, while the remaining software types contribute only isolated instances. Overall, Missing bugs suggest less diversity in bug operations and greater concentration across software types, with omitted data checks as the most common pattern at the software-type level.

In contrast, Figure 2.13b shows that **Incorrect** bugs exhibit a broader and more heterogeneous distribution in the retained corpus. Six distinct operations are observed, with CALL-related mistakes emerging as the most frequent mechanism (26.7%, 4/15). *Libraries* form the main concentration point (5/15, 33.3%) while spanning multiple operations, including CALL, DECLARE, EVALUATE, and DEREFERENCE; the remaining cases are distributed across *web applications*, *extensions*, *utilities*, and *servers*. Overall, Incorrect bugs suggest greater diversity in bug operations and less concentration across software types.

AQ2. Where do security bugs reside in software?

> ***Intent:*** *Examine where bugs concentrate within software functional areas by root cause.*

Figure 2.14a summarizes the distribution of security bugs by root cause across software functional areas in the retained corpus and shows that **Missing** bugs concentrate in data processing components. *File Processing* accounts for the largest share of Missing cases (37.5%, 15/40), followed by *String Processing* (30.0%, 12/40) and *Number Processing* (10.0%, 4/40). Operationally,



(a) Missing *Software Type*×*Operation* Heatmap

(b) Incorrect *Software Type*×*Operation* Heatmap

Figure 2.13: Distribution of bug operation across software types by cause. Each heatmap reports the concentration of operations by software type, contrasting Missing and Incorrect bugs.

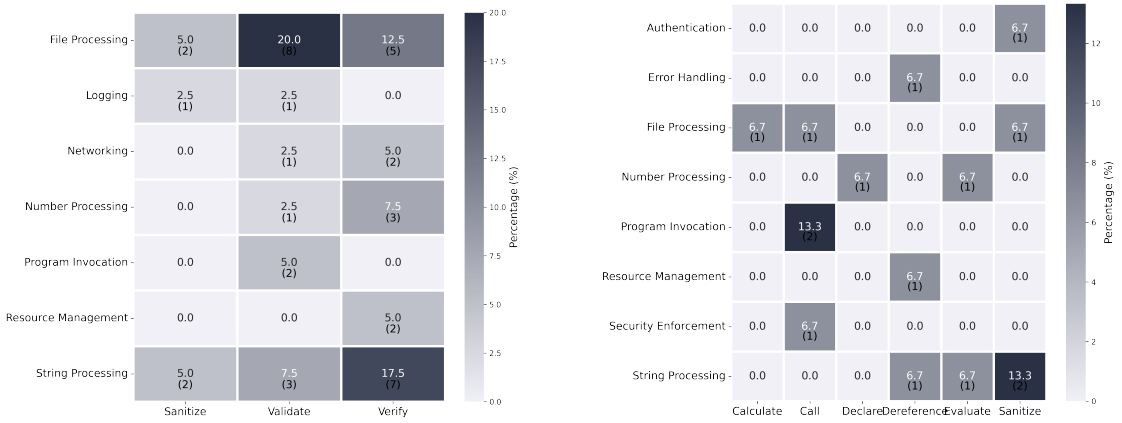
VERIFY accounts for the largest share (47.5%, 19/40) and appears across most functional areas except *Logging* and *Program Invocation*. VALIDATE is similarly prevalent (40.0%, 16/40), concentrating almost exclusively in processing domains—*File*, *String*, and *Number Processing*. SANITIZE plays a secondary role (12.5%, 5/40) in *String Processing*, *File Processing*, and *Logging* components. Overall, Missing bugs suggest less diversity in functional area, with omitted data checks concentrated in components that handle and transform inputs.

In contrast, Figure 2.14b shows that **Incorrect** bugs are more dispersed in the retained corpus. They cover eight functional areas, including *Authentication*, *Error Handling*, and *Security Enforcement*, which do not appear in the Missing cases, while data processing components still account for most cases (60.0%, 9/15), with *String Processing* contributing the largest share (4/15). The distribution of operations is broader, with CALL and SANITIZE as the most frequent mechanisms (26.7%, 4/15 each), half of these occurring in *Program Invocation* and *String Processing*. DEREFERENCE contributes three cases and EVALUATE two, while CALCULATE and DECLARE appear once. Overall, Incorrect bugs suggest greater diversity in functional area, with data processing components still accounting for most observations.

AQ3. How do security bugs vary by programming language?

> ***Intent:*** *Examine how language shapes concentration and diversity of bugs by root cause.*

Figure 2.15a summarizes the distribution of security bugs by root cause across programming languages in the retained corpus and shows that **Missing** bugs span all languages in the dataset but are concentrated in *PHP*, which accounts for 62.5% of Missing cases (25/40). *C* contributes a second visible concentration with 17.5% (7/40), while *Python* contributes three cases, *JavaScript* two, and *Java*, *Go*, and *C++* one each. At the operation level, *PHP* concentrates missing VALIDATE and SANITIZE, whereas *C* shows a distinct concentration of missing VERIFY (15.0%, 6/40). Overall, Missing bugs suggest less diversity across programming languages, with a language-specific

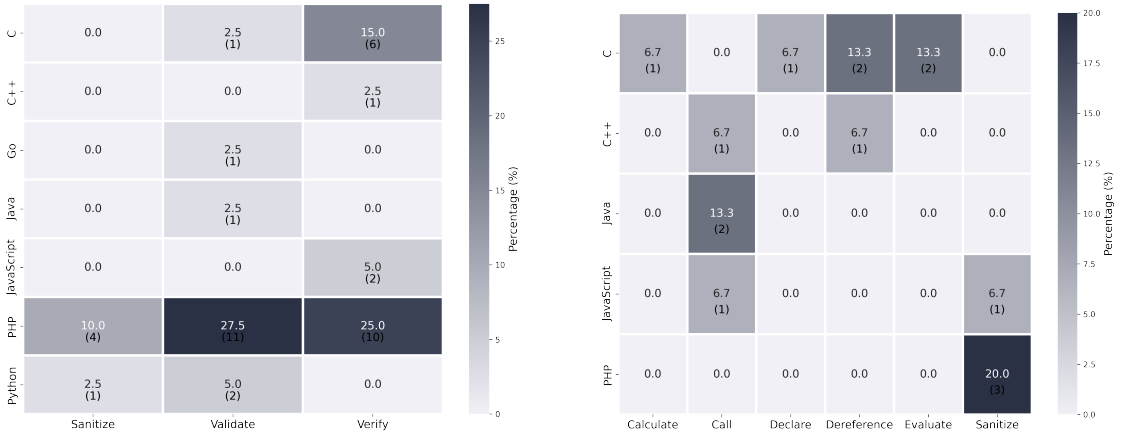


(a) Missing *Functional Area*×*Operation* Label (b) Incorrect *Functional Area*×*Operation* Label

Figure 2.14: Distribution of bug operation across functional areas by cause. Each heatmap reports the concentration of operations by functional area, contrasting Missing and Incorrect bugs.

concentration of omitted data checks.

In contrast, Figure 2.15b shows that **Incorrect** bugs are distributed more selectively across programming languages in the retained corpus. They appear in five languages, with *C* contributing the largest share (40.0%, 6/15), followed by *PHP* with three cases and *JavaScript*, *Java*, and *C++* with two cases each. At the operation level, incorrect SANITIZE is limited to *PHP* and *JavaScript*, while incorrect CALL appears only in *JavaScript*, *Java*, and *C++*, reflecting mistakes in API binds. Incorrect CALCULATE, DECLARE, and EVALUATE concentrate in *C*, and incorrect DEREFERENCE is split between *C* and *C++*. Overall, Incorrect bugs also suggest language-specific clustering in *C*, with higher-level languages exhibiting API misuse.



(a) Missing *Language*×*Operation* Label (b) Incorrect *Language*×*Operation* Label

Figure 2.15: Distributions of bug operations across progr. languages by cause. Each heatmap reports the concentration of operations by language, contrasting Missing and Incorrect bugs.

Table 2.2: Frequent tuples of the underlying bug profiles.

Behavior	Operation Cat.	Software type¶	Functional Cat.	Language	Support*
Missing	Data Check	Web App	Data Processing	PHP	55.0% (22)
Missing	Data Check	Web App	Resource Mang.	PHP	5.0% (2) †
Missing	Data Check	Library	Data Processing	JavaScript	5.0% (2) †
Missing	Data Check	Framework	Data Processing	C	5.0% (2) †
Incorrect	Data Type	—	Data Processing	C	26.7% (4) †
Incorrect	Data Check	—	Data Processing	PHP	13.3% (2) †
Incorrect	Dereference	—	Error Handling	C	6.7% (1) †

**Percentages are computed within behavior class (Missing: n=40, Incorrect: n=15).*

† Count <5; percentages based on small counts should be interpreted cautiously.

¶ Omitted for ***Incorrect*** *because these are fewer and more sparsely distributed.*

Table 2.2 highlights the recurring combinations identified as security bug profiles in the corpus. To support synthesis across AQ1–AQ3, fine-grained operation and functional-area labels are grouped into categories that capture recurring patterns to avoid fragmentation (see Appendix A.2.1). These recurring combinations show that the corpus contains a small number of profiles separated by cause. **(M1)** Missing bugs are concentrated in omitted *Data Checks* in *Data Processing* components of *PHP Web Applications* (n=22), reflecting the repeated absence of validation, verification, and sanitization logic in externally exposed, data-intensive code paths. **(I1)** Incorrect cases show a distinct but smaller pattern: *Data Type* bugs in *Data Processing* components written in *C* (n=4), corresponding to mistakes tied to type-related operations across heterogeneous software types. The remaining observations are distributed across additional languages and functional areas, but with insufficient recurrence to support additional profiles.

> ***Finding 1.*** *Software security bugs in the corpus can be profiled through recurring combinations of root cause and software context.* Data Processing *components are the main locus of bugs for both causes.* ***Missing*** *bugs are dominated by omitted* Data Checks*, most prominently in* PHP Web Applications*, whereas* ***Incorrect*** *bugs are primarily driven by* Data Type *mistakes recurrent in* C*.*

2.5.3 Manifestation Chains of Software Security Bugs

Building on the identified bug profiles, this subsection addresses the second part of RQ1 by examining how the security bugs in the corpus propagate through error and failure outcomes. Figure 2.16 provides an overview of the main manifestation chains. The root-cause layer, represented by *cause* and *operation*, follows the patterns identified in Section 2.5.2, with most flow originating from *Missing* cause and passing through a small set of operation categories. These converge primarily on *Semantic Corruption* (29, 52.7%), followed by *State Corruption* (15, 27.3%) and *Authority Violation* (11, 20.0%) errors, while resulting failures are concentrated in *Code Execution* (50, 90.9%), with the remaining cases mapped to *Denial of Service* (2, 3.6%), *Information Exposure* (2, 3.6%), and *Tampering* (1, 1.8%).

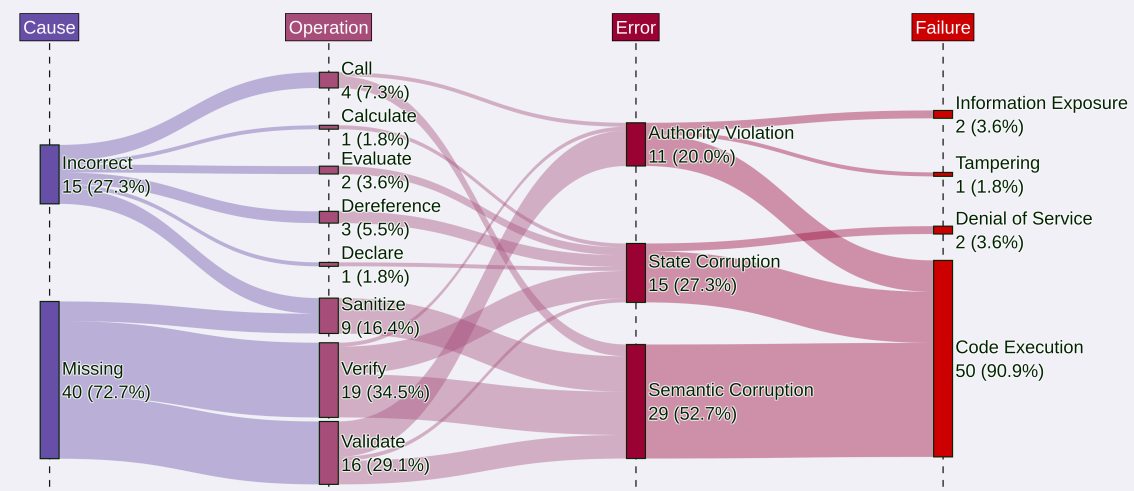


Figure 2.16: Sankey-style overview of the manifestation chains linking bug cause and operation to propagated errors types and failure outcomes.

To ground the analysis of manifestation chains, Table 2.3 first reports the distributions of errors and failures by cause. Under **Missing** cause, errors are concentrated in *Semantic Corruption* (22 cases, 75.9%), followed by *Authority Violation* (10 cases, 90.9%), while *State Corruption* contributes a smaller share (8 cases, 53.3%). These errors most often culminate in *Code Execution* failures (37/50, 74.0%), with only isolated instances of *Information Exposure* (2 cases) and *Tampering* (1 case), and no observed *Denial of Service*. **Incorrect** cause is distributed more evenly across error categories: *Semantic Corruption* and *State Corruption* each contribute seven cases (24.1% and 46.7%, respectively), whereas *Authority Violation* appears only once. Although *Code Execution* remains the most frequent failure outcome (13/15, 86.7%), Incorrect cause accounts for all observed *Denial of Service* failures (2 cases).

Table 2.4 presents the manifestations associated with the dominant security bug profiles and shows that these most often culminate in *Code Execution*, but through distinct error propagation paths. For the dominant **Missing** profile **M1**, the primary pathway proceeds through *Semantic Corruption* to *Code Execution* (15 cases, 68.2%), reflecting violations of interface semantics caused by omitted VALIDATE, VERIFY, and SANITIZE operations in externally exposed, data-processing code. A secondary pathway passes through *Authority Violation* to *Code Execution* (6 cases, 27.3%), with a single instance of *Authority Violation* resulting in *Information Exposure* (1

Table 2.3: Baseline disttributions of errors and failures by cause.

Cause	**Errors**			**Failures**			
	Semantic Corruption	*Authority Violation*	*State Corruption*	*Code Execution*	*Denial of Service*	*Information Exposure*	*Tampering*
Missing	75.9% (22)	90.9% (10)	53.3% (8)	74.0% (37)	—	100% (2)	100% (1)
Incorrect	24.1% (7)	9.1% (1)	46.7% (7)	26.0% (13)	100% (2)	—	—

Note: *Percentages computed column-wise (within each error/failure type).*

Table 2.4: Dominant error–failure manifestations enabled by canonical security bug profiles.

Profile	Error	Failure	Support*
M1	Semantic Corruption	Code Execution	68.2% (15)
	Authority Violation	Code Execution	27.3% (6)
	Authority Violation	Information Exposure	4.5% (1) †
I1	State Corruption	Code Execution	75.0% (3)
	State Corruption	Denial of Service	25.0% (1) †

**Percentages are computed within each profile (M1: n=22, I1: n=4).*

† Single instances are treated as coverage rather than dominant patterns.

case, 4.5%). By contrast, the dominant **Incorrect** profile **I1** propagates exclusively through *State Corruption*. Most cases lead to *Code Execution* (3 cases, 75.0%), while a smaller fraction results in *Denial of Service* (1 case, 25.0%), suggesting that low-level mistakes affecting internal program state can either compromise execution integrity or disrupt service availability. No other error–failure pairs appear with sufficient recurrence to support additional manifestation patterns.

Finding 2. *Dominant security bug profiles in the corpus converge on similar failures but propagate through distinct errors. Both Missing and Incorrect profiles most frequently culminate in* Code Execution*; however, Missing cases reach this outcome via* Semantic Corruption *or* Authority Violation*, whereas Incorrect cases propagate exclusively via* State Corruption*. Thus, root cause and software context shape error propagation, even when the resulting security failures align.*

2.6 Discussion

This section situates the empirical findings within a broader analytical and practical context. Section 2.6.1 interprets the observed bug profiles and manifestation patterns. Section 2.6.2 outlines implications for security research, automation, and defensive practice. Section 2.6.3 examines methodological limitations and validity constraints associated with reconstruction, labeling, and corpus selection. Finally, Section 2.6.4 identifies directions for extending coverage, refining labels, and improving scalability.

2.6.1 Interpretation of results

This chapter yields two main findings that clarify how security bugs recur across software and how they manifest in the analyzed corpus. Although the corpus is limited, the reconstruction workflow recovered bug $\rightarrow$ error $\rightarrow$ failure chains from associated artifacts, and the labeling schema was sufficiently expressive to capture root causes, software context, and manifestations. The first finding shows that recurring software security bugs can be synthesized compactly into a small set of profiles, represented as tuples of *cause*, *operation*, *software type*, *functional area*, and *programming*

language. The second finding shows that in these profiles, bugs propagate through distinct error–failure manifestation paths even when final outcomes converge on the same failure, most notably *Code Execution*. These findings show that reconstruction and labeling can recover informative structure from vulnerability cases, while profiling provides an analytical lens for interpreting security bugs in software context.

Software type. In the corpus, software type is associated with distinct root causes, indicating where untrusted data enters the system and where security policy must be enforced. Web applications concentrate externally reachable data-handling paths—file uploads, request parameters, configuration writes, and dynamic content generation, making validation, sanitization, and verification indispensable at these boundaries. This helps explain why Missing *Data Checks* are concentrated in PHP web applications: when enforcement is manual and interleaved with application logic, omissions become a common mistake. This pattern appears in cases involving missing file validation (*CVE-2020-18879*), unchecked configuration paths (*CVE-2020-18185*), unsafe XML parsing defaults (*CVE-2023-46502*), and untyped request parameters (*CVE-2018-13447*).

Functional area. In the corpus, functional area is associated with distinct root causes and highlights where security bugs concentrate, with Data Processing components accounting for the largest share. This is where inputs are parsed, transformed, or interpreted—whether as files, strings, numbers, or structured documents—making assumptions about format, size, lifetime, and trust especially fragile. For Missing bugs, these components often omit defensive checks, as in bounds validation in file readers (*CVE-2019-10878*), path canonicalization during extraction (*CVE-2022-44900*), or input typing in request handlers (*CVE-2018-13450*). For Incorrect bugs, the same areas show implementation mistakes that corrupt internal state, including the use of an invalid pointer after re-entrant callbacks (*CVE-2020-20703*), unsafe integer arithmetic during sizing (*CVE-2020-19692*), and pointer invalidation by stack unwinding during error handling (*CVE-2018-11499*). In the corpus, these patterns suggest that Data Processing components are a recurring locus for both omitted checks and implementation mistakes.

Vulnerability chains. Our second finding highlights a limitation in how vulnerabilities are commonly described and classified. CVE descriptions often emphasize the manifestation side of a vulnerability, especially its consequence and impact, while leaving the underlying bug underspecified. This emphasis is reinforced by NVD CWE assignments, which frequently foreground the manifested weakness; for example, identifiers such as **CWE-79**, **CWE-89**, **CWE-787**, **CWE-22**, and **CWE-434** strongly cue the consequence, sometimes alongside broader root-cause phrasing such as “Improper Neutralization.” Because CWE labels often combine the underlying cause and the consequence in a single entry, the elements of a vulnerability chain are not always clearly separated. As a result, vulnerabilities with similar manifestations may in practice arise through

different bug–error pathways. The reconstruction and labeling workflow helps recover that structure. In the analyzed corpus, even when cases converge on the same failure, most often *Code Execution*, they do so through different pathways: Missing bugs tend to induce *Semantic Corruption* or *Authority Violation* errors, whereas Incorrect bugs propagate through *State Corruption* errors. This shows why reconstructing vulnerability chains matters in practice: detection and repair must ultimately address the underlying bug.

2.6.2 Implications for reporting, research, and practice

The findings show that relating root causes to software context and reconstructing vulnerability chains enables a more precise account of why security bugs arise, where they concentrate, and how they manifest. This supports improvements in vulnerability reporting, closer integration of root cause and software context in security research, and more targeted approaches to automation and defensive practice. The following discussion develops these implications in turn.

For vulnerability reporting. A central implication of this chapter is that current vulnerability records often lack the structured information needed to support reliable automation, reconstruction, and bug profiling at scale. In practice, this means that NVD records would benefit from more complete and better maintained metadata, including precise identification of the main vulnerable product in the CPE, a field indicating whether that product is open source or private, a direct link to the repository when it is open source, consistent CVSS assessments, up-to-date CWE mappings, and direct references to the fixing commit. CVE descriptions also need improvement: many records contain grammatical errors, do not follow a consistent template, or omit key details about the root cause, the weakness, and the impact. Without such improvements in reporting, vulnerability datasets will remain small, limiting their value for empirical analysis.

For security research. The results suggest that root cause and software context need to be analyzed together to explain both where security bugs concentrate and how they manifest. Existing taxonomies already capture parts of this space: bug categories are represented in the Bugs Framework, while software dimensions such as *functional area* and *programming language* also appear in CWE, although unevenly and not as a combined profiling lens. Doing so would provide a clearer empirical basis for understanding why certain bugs arise in some parts of software and how their manifestations affect different parts of a system. Security research should therefore bridge this gap by relating software context more closely to root causes and manifestations, rather than treating these dimensions in isolation.

For automation and tooling. The resulting data provides an initial ground truth for evaluating whether automated methods recover underlying bugs or drift toward their manifestations. It also highlights a harder target for automation: many cases involve **Missing** cause, where the bug is not an incorrect statement but omitted logic that may have no single explicit location in code. In such cases, automation must infer that a relevant program action is absent, rather than merely flag

code that appears wrong. The findings further suggest that tooling should move beyond single-line reporting toward representations that capture and explain vulnerability chains in code and at runtime, since the path from bug to failure may involve multiple intermediate erroneous states and span multiple locations.

For practitioners and defenders. The findings point to recurring security hotspots in software systems. Data-processing components—such as parsers, upload handlers, serializers, and configuration loaders—emerge as recurring vulnerable loci, where both Missing and Incorrect bugs concentrate. Because the reconstruction is grounded in fixing patches, these implications are not only abstract categories but correspond to concrete repair actions: Missing bugs benefit most from validation, verification, and sanitization, whereas Incorrect bugs require correctness in arithmetic, memory management, and control flow. Practitioners should therefore not only prioritize these hotspots, but also implement preventive measures at those locations that interrupt bug propagation before it reaches security-critical operations.

2.6.3 Threats to validity and limitations

The main limitation of this study concerns external validity, as the findings are based on a small corpus and may not generalize to the broader vulnerability population. This corpus depends heavily on the availability and quality of the public artifacts needed for reconstruction, especially sufficiently detailed CVE descriptions and direct links to fixing commits. The workflow relies on expert and interpretive judgment. Reconstruction requires manually identifying the bug in the fixing commit, while labeling is likewise manual and applies a custom schema to the available artifacts. Together, these constraints limit scalability and introduce subjectivity despite the use of documented heuristics and consistency checks.

External validity. The main limitation of this study concerns generalization. The observed distributions and recurring patterns are conditional on a small, highly filtered corpus and should not be interpreted as representative of the broader vulnerability population. Of the 4,324 Critical-severity CVEs initially considered, screening for grammatical acceptability and single-sentence structure removed more than half, and subsequent reporting template and entity extraction requirements reduced the candidate set to 451 CVEs. Reliance on NVD references further narrowed the pool: only 66 CVEs included usable links to commits or issues that enabled patch analysis. As a result, the final corpus is shaped not only by the underlying vulnerabilities, but also by reporting quality, repository traceability, and patch availability, and may therefore underrepresent noisy, poorly documented, or non-standard cases.

Patch noise. Real-world security patches often include non-security refactoring, test files, or repeated fix edits within the same commit, which can complicate bug identification. To keep reconstruction attributable, the analysis is therefore limited to cases—or to patch regions—where

the bug can be assigned with confidence to the reported vulnerability. For example, *CVE-2025-25977* involves a patch touching 123 files, most of them tests, and 11 code files with repeated edits for unsafe property access. Only the edits tied to the constructor named in the CVE description and the variable identified in the issue report[6] were considered for labeling. This keeps labeling focused on the most directly attributable bug while excluding cases where patch noise prevents confident identification.

CWE mismatch. In this study, the weakness described in the record is first mapped to a CWE identifier, checked against the primary CWE reported in NVD metadata for alignment. Public metadata can still be inaccurate, including suboptimal choices among related CWEs or assignments that do not match the weakness supported by the evidence. For example, in *CVE-2023-46932*, NVD assigns `CWE-787` (Out-of-bounds Write), whereas the associated issue[7] reports a heap-buffer-overflow on a *read* of size 1. Accordingly, the manifestation was labeled as `CWE-125`. Similar discrepancies were observed in *CVE-2020-20276*, *CVE-2020-19692*, and *CVE-2020-20703*. To reduce this threat, manifestation labels were validated against the available evidence in issue reports and fix commit messages.

Label ambiguity. Classifying bugs involves cases where multiple interpretations are plausible. This occurs when a bug can be described at different levels of abstraction or spans multiple operations. For example, in *CVE-2020-25911*, untrusted XML is parsed without disabling external entity resolution, which may be labeled either as Incorrect DECLARE (an insecure parser configuration) or as Missing VALIDATE (an absent safety check). To reduce inconsistency, the analysis used BF operations, CWE definitions, and documented disambiguation heuristics. When an operation was not directly observable in the vulnerable code, the closest CWE was used to interpret the case and align it with a BF operation.

2.6.4 Future work

This study provides an initial foundation for profiling security bugs and reconstructing vulnerability chains from public artifacts. Future work can extend this foundation by improving the scalability and robustness of the reconstruction and labeling workflow through AI-based assistance, expanding corpus coverage, and refining the labeling schema. Because the present dataset was created manually, it also provides a basis for evaluating how reliably such methods approximate expert judgments. Together, these directions address current limitations in dataset scope, label ambiguity, and manual effort.

Scalability and automation. Future work can reduce the corpus attrition identified in the limitations and make larger-scale analysis more feasible by extending the present methodology with AI-based systems to execute the manual parts of the reconstruction and labeling workflow. Large

[6] https://github.com/canvg/canvg/issues/1749
[7] https://github.com/gpac/gpac/issues/2669

language models could help compensate for issues in CVE records by normalizing irregular descriptions, recovering vulnerability details from incomplete reports, reconciling metadata across available artifacts, and improving traceability from products to repositories. Agentic workflows could also support reconstruction by scanning repository histories for candidate fixing commits, locating the bug within long or noisy patches, and producing structured explanations for downstream classification. Such assistance could make the workflow more scalable while still requiring expert validation, shifting the human role toward reviewing AI-generated cases. As the dataset grows, the manually reconstructed and labeled cases could also serve as a benchmark for evaluating how effectively AI-based methods approximate expert-level performance in this task.

Sample diversity. A natural next step is to expand the dataset beyond the current quality filters, particularly the focus on Critical vulnerabilities. Incorporating Medium- and High-severity CVEs would broaden the sample and allow the profiles identified here to be tested across a wider severity range. This would support analysis of which profile attributes—such as cause, operation, or software context—are most strongly associated with severity. Broader coverage may reveal additional recurring patterns while strengthening confidence in the profiles already observed.

Richer context. Future work can refine the contextual attributes used in profiling without sacrificing interpretability. One promising direction is to extend the *functional area* attribute to distinguish, for example, inbound from outbound data-processing components, or externally exposed APIs from internal subsystems. This may clarify whether some contexts are more strongly associated with bug introduction, while others are more strongly associated with failure realization. Whether these refinements improve profile separability and explanatory power without overfragmenting the schema remains an empirical question.

2.7 Summary

Objective. Vulnerability reporting often describes vulnerabilities through their manifestations rather than through the bugs that cause them. CVE descriptions typically emphasize errors and failure outcomes, while the root cause is implicit or omitted, so recovering it usually requires patch analysis. Although initiatives such as MITRE's Root Cause Mapping and NIST's BF highlight the need for root cause characterization, their representations of weaknesses include both causes and consequences, which BF makes explicit. This chapter reconstructs vulnerabilities from public artifacts as *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chains, represents each case in terms of *root cause*, *software context*, and *manifestation*, and analyzes recurring bugs across software contexts as *bug profiles*.

Method. To support reliable reconstruction, the methodology begins by selecting only consistent and sufficiently enriched CVE records. It then restricts the corpus to comparable vulnerabilities, namely *low-contingency* cases with *critical* impact and *unchanged scope*, so activation and consequences remain within the same authority boundary, and to open-source *application* components

with traceable repositories and accessible patches so the bug can be recovered from code. Reconstruction proceeds by extracting manifestation details from the CVE description—specifically *weakness* and *impact*—and tracing backward through the corresponding patch to infer the enabling *bug*. The reconstructed cases are then labeled in terms of *root cause*, *software context*, *error*, and *failure*, yielding bug profiles represented across five axes—*cause*, *operation*, *functional area*, *software type*, and *programming language*.

Findings. In the analyzed corpus, *Data Processing* components account for the largest share of bugs for both causes: *Missing* bugs concentrate around omitted *Data Checks*, most prominently in *PHP Web Applications*, whereas *Incorrect* bugs concentrate around *Data Type* mistakes recurrent in *C*. This shows that profiling helps interpret how security bugs recur in software. The manifestations of these bug profiles converge on similar failures but propagate through distinct errors, most notably in cases culminating in *Code Execution*: *Missing* bugs propagate through *Semantic Corruption* or *Authority Violation*, whereas *Incorrect* bugs propagate through *State Corruption*. This shows that reconstruction and labeling recover informative structure from vulnerability cases. Together, these findings show that bug profiling and vulnerability reconstruction can explain why security bugs arise, where they concentrate, and how they manifest.

Implications. This perspective can help identify security hotspots and supports tooling that moves beyond single locations toward representations of the full vulnerability chain. *Missing* cases, however, are inherently harder to localize and automate because omitted logic may not correspond to an explicit code location. These insights should be considered with caution because the corpus is small, limiting external validity. Scaling the approach remains difficult because the workflow depends on data quality and manual effort. Better vulnerability reporting is therefore a key requirement for broader empirical use, particularly more complete metadata, clearer descriptions, and direct links to code artifacts. AI-based methods could handle the manual parts to scale the workflow, with the current dataset providing a benchmark for evaluating how closely they approximate expert judgment.

Chapter 3

Crafting a Vulnerability Benchmark

3.1 Introduction

This chapter extends the vulnerability representation introduced in Chapter 2 by grounding the manifestation side in reproduced crash reports. Chapter 2 characterized vulnerabilities through their bug → error → failure structure, showing that similar failure outcomes can arise from different bugs and propagate through different errors. This chapter turns that representation into a benchmark by reproducing vulnerabilities, localizing the bug from the fixing patch, and recording the error from the crash report. In this way, the benchmark preserves both the bug and the manifestation as distinct elements of the vulnerability. The remainder of this introduction motivates the gap 3.1.1, outlines the benchmark construction approach and scope 3.1.2, states RQ2 and contributions 3.1.3, and positions the chapter in the broader thesis context 3.1.4.

3.1.1 Motivation and gap

Evaluating vulnerability detection and repair techniques depends on benchmarks that provide reliable ground truth. In practice, however, publicly disclosed vulnerabilities rarely translate directly into instances ready for evaluation: reports often describe how a vulnerability manifests (e.g., a crash or memory violation) while omitting the underlying bug, the historical build context, or the artifacts required for deterministic reproduction. As a result, evaluations tend to rely on datasets representing the manifestation (error or failure) rather than the underlying bug that tools are intended to detect.

This gap matters because a benchmark implicitly defines what constitutes correct detection. If ground truth reflects only error locations or a coarse CWE category, tools may be rewarded for matching manifestations rather than identifying the underlying bug. Chapter 2 showed that vulnerabilities follow a bug → error → failure chain, and that focusing solely on the final manifestation obscures how bugs arise and propagate. This chapter, therefore, addresses the challenge of transforming real-world vulnerability disclosures into a benchmark in which each instance is reproducible from the vulnerable version and grounded in the underlying bug.

Reproducibility further degrades over time: dependencies disappear, build systems evolve, and repositories become inaccessible. At the same time, disclosures often provide incomplete inputs and operational details, and the linkage between reports and code history remains ambiguous. These factors contribute to the compression of diverse bugs into a small set of manifestation categories. Prior work addresses parts of this problem—synthetic datasets scale but provide simplified vulnerabilities, while curated benchmarks and fuzzing infrastructures treat the observed errors as ground truth, leaving the link between reproduced crashes and the underlying bug undefined.

3.1.2 Benchmark construction approach and scope

Building on the representation introduced in Chapter 2, this chapter develops an end-to-end benchmark construction pipeline that transforms fuzzing-derived disclosures into executable, validated benchmark instances. The pipeline extracts OSS-Fuzz vulnerabilities from OSV, links them to CVEs using multiple evidence signals, reconstructs historical build contexts, reproduces crashes by replaying fuzzing test cases, and localizes both the error location and the bug location using crash reports and fixing patches. Only vulnerabilities that can be rebuilt, triggered in the introducing revision, and validated in the fixing revision are retained. Reproduced cases are validated by checking that the generated crash reports match the OSS-Fuzz reports and that the localized bug is consistent with the fixing patch.

Each benchmark entry records two elements of the vulnerability: the error and the bug. The *error* is derived from crash reports (crash type, operation, crash-mapped CWE) and captures the program operation at the location where the failure becomes observable. The *bug* is derived from patch artifacts through root cause localization and labeling, and captures the improper program operations. Root cause localization is semi-automatic, using proximity heuristics between stack trace locations and the modified lines in the patch diff, followed by manual confirmation. Root cause labeling follows the schema introduced in Chapter 2. This representation supports evaluation settings that distinguish between detecting the underlying bug and detecting its manifestation.

The scope of this chapter is grounded in what can be reconstructed and validated from publicly available fuzzing artifacts. As a result, the reproduced cases are biased toward fuzzing-discovered memory-safety vulnerabilities in C/C++ programs. Root cause labeling also involves interpretation, as common taxonomies can conflate underlying causes with their manifestations. These constraints limit generality but provide a benchmark that is executable, reproducible, and suitable for controlled evaluation in subsequent chapters.

3.1.3 Research questions and contributions

This chapter translates the representation introduced in Chapter 2 into a benchmark design question: **RQ2. Which vulnerability attributes are required in a benchmark for evaluation?**

> A benchmark defines what constitutes correct detection. If it retains only the manifestation, tools may be evaluated based on agreement with the manifestation rather than identification of the underlying bug. If it lacks executable artifacts, results cannot be reproduced. Because vulnerabilities follow a bug → error → failure chain, the benchmark must preserve the attributes that distinguish the bug from its manifestation. Otherwise, evaluation outcomes may reflect incidental agreement rather than the detection of the bug.

To answer this question, the chapter makes the following contributions:

- **A benchmark of reproduced crashes from public vulnerability records.** Using OSV and NVD disclosures, we construct an executable core of 30 manually validated vulnerabilities that reproduce the reported crashes, from 69 OSV-CVE links and 40 buildable cases.
- **A benchmark representation that records both bug and error.** Each reproduced case records the vulnerable version, the crashing input, the fixing commit, the bug location, and the error location, treating the bug and the error as distinct elements of the vulnerability.
- **An empirical analysis of the relation between bug and error locations.** The analysis shows that the bug location and the error location frequently diverge across functions and files (13 of 30 cases), and can remain separated even within a function (11 of 17 same-function cases).
- **An empirical characterization of operations at bug and error locations.** At the error location, operations correspond to memory usage (WRITE and READ), while bug operations span 8 distinct types—dominated by VERIFY (21 of 30 cases, 70.0%) under a *Missing* cause (20 of 30 cases, 66.7%)
- **Benchmark requirements for reliable evaluation.** From these results, the chapter identifies the information a benchmark must retain: reproducible cases tied to a vulnerable version and fixing commit, explicit recording of bug and error locations, and labels that characterize the bug in terms of cause and operation.

These findings are bounded by a small corpus of fuzzing-amenable C/C++ projects and may not generalize to the broader vulnerability population. Extending the pipeline to additional vulnerability and project types is the most direct path toward broader generalizability. The reproduced cases and their manually validated labels nevertheless provide an executable benchmark with explicit bug and error locations, offering a foundation for evaluations that distinguish between detecting the root cause and detecting its manifestation—a distinction that benchmarks retaining only the error side cannot support.

3.1.4 Publications context and roadmap

This chapter builds on two prior lines of work that exposed both the need for and the challenges of constructing benchmarks from public vulnerability artifacts. In Maestro [190], we developed

an evaluation platform for automatic program repair on vulnerabilities. Its scope, however, was limited to a small set of projects because reproducible vulnerability benchmarks were largely unavailable. This experience highlighted that the main limitation was not tooling, but the lack of instances that can be rebuilt, triggered, and linked to their fixing commits. This chapter addresses that limitation by defining a process to reconstruct such instances from public disclosures.

A subsequent study on replicability and deployment reliability of deep neural networks [191] provided complementary methodological experience in assembling benchmarks from heterogeneous public artifacts. Although the domain differed, the effort highlighted practical requirements for experimental rigor, including transparent artifact packaging, modular tooling, environment setup, and explicit handling of variability. These principles inform the design of the reproduction pipeline, validation workflow, and artifact management used in this chapter, ensuring that the resulting benchmark is reproducible and extensible.

Roadmap. The remainder of the chapter is organized as follows. Section 3.2 introduces background and the motivating gap. Section 3.3 discusses related work and how existing datasets and studies fall short of ground truth recording the bug. Section 3.4 details the benchmark construction methodology. Section 3.5 reports reconstruction yield, bug–error separation, and the analysis of operations at the two locations. Section 3.6 discusses implications, limitations, and future directions. Finally, Section 3.7 summarizes the chapter's contributions.

3.2 Background

This section introduces the core concepts required to understand the remainder of the chapter. Section 3.2.1 restates the defect–error–failure distinction that separates originating defects from manifested failures. Section 3.2.2 defines the properties of reproducible, version-anchored vulnerability instances. Section 3.2.3 introduces dynamic analysis and fuzzing as mechanisms for generating executable crash artifacts. Section 3.2.4 summarizes fault localization techniques that infer fault-relevant code regions from execution evidence. Together, these concepts provide the necessary basis for the content presented in subsequent sections.

3.2.1 Conceptual Basis: The Defect–Error–Failure Distinction

This chapter operationalizes the causal model introduced in Chapter 2. We briefly restate the elements required for benchmark construction. Across standards and taxonomies, a *vulnerability* is consistently framed as a weakness that can be exploited to compromise confidentiality, integrity, or availability (CIA) [218, 39, 37]. From a dependability perspective, Avizienis *et al.* [10] formalize this notion through the fault–error–failure chain: a *fault* (defect) is introduced during development, its activation produces an incorrect system state (an *error*), and this state may propagate into an externally observable *failure*. In this view, a vulnerability corresponds to a security-relevant fault whose activation can violate CIA properties.

Chapter 2 emphasized that failures often occur at locations distinct from the originating defect. A memory corruption may manifest as a crash in one function while the underlying defect resides elsewhere; similarly, multiple defect mechanisms may produce the same observable failure class. This separation between *bug site* (defect origin) and *error site* (failure manifestation) is central to evaluation: detection at the manifestation level does not necessarily imply identification of the underlying mechanism. Standard representation frameworks further reflect this distinction at different abstraction levels. CWE provides structural weakness categories [35], CVE assigns identifiers to disclosed vulnerability instances [48], and CVSS quantifies impact [155]. The NIST Bugs Framework (BF) [16, 18] complements these by modeling weaknesses through causal execution semantics, linking improper operations to resulting failures. Together, these frameworks provide the representational vocabulary needed to distinguish manifestations from defect-level labels.

In this chapter, the defect–error–failure model is used operationally: benchmark instances preserve both the failure manifestation and the originating defect, enabling evaluation to differentiate between manifestation detection and bug identification.

3.2.2 Reproducible Vulnerability Instances

Empirical evaluation of vulnerability detection and repair techniques depends on instances that can be reconstructed and exercised deterministically. In practice, publicly disclosed vulnerabilities often lack the artifacts required for reliable reproduction: precise version information [32], build configurations [154], triggering inputs [46], or explicit linkage to fixing commits [171]. Without such elements, evaluation reduces to consequence-level labeling or synthetic approximation. Reproducibility, therefore, serves as a prerequisite for meaningful benchmarking, ensuring that reported failures correspond to verifiable program behavior rather than metadata descriptions.

A *reproducible vulnerability instance* minimally requires report-tied information sufficient to complete the four-stage reproduction workflow: (i) reports and references that specify the vulnerable software version(s) and enable obtaining the corresponding source/binary; (ii) instructions to set up the required operating system environment for that version; (iii) build/installation guidance (including configuration options and required libraries) to compile and install the software; and (iv) a PoC and verification criteria to trigger and confirm the vulnerability [168]. Continuous fuzzing infrastructure [202] and version control systems [75] provide the artifacts necessary to approximate these conditions: minimized-crashing inputs, sanitizer reports, and commit histories that bound the introduction and remediation of vulnerabilities. Together, these artifacts ground vulnerabilities in executable program states rather than solely in textual disclosures.

For benchmark construction, the focus must be on enabling vulnerability reproduction rather than simply recording vulnerability information. Reproduction is particularly likely to fail when the vulnerable version, PoC, or trigger method are missing, while other omissions (e.g., installation defaults or OS details) can sometimes be recovered through common assumptions [168]. Each benchmark instance must therefore be version-anchored (e.g., via commit hashes identifying the vulnerable revision), include the corresponding source code, and provide a PoC with the exact execution procedure required to deterministically trigger the bug. In addition, reproducible datasets

require a rebuildable pipeline that captures the build environment, dependencies, and compilation configuration, since dependency drift and missing upstream resources can prevent historical recompilation [154]. These elements ensure that each instance can be reproduced in a controlled, repeatable manner.

3.2.3 Dynamic Analysis and Fuzzing

Dynamic analysis studies program behavior during execution. By instrumenting a program and observing its runtime state, dynamic techniques collect execution traces, coverage information, or fault signals that reflect concrete program behavior [108, 22]. Unlike purely static reasoning, dynamic analysis operates on actual program executions and therefore provides precise evidence about triggered states and failures. In the context of vulnerability research, dynamic analysis supplies the executable artifacts—crashes, traces, and inputs—that ground empirical evaluation.

Dynamic analysis techniques vary in the semantic insight they employ [212, 11]. Two widely studied approaches are:

- *Dynamic Concrete Execution (DCE)*: executes a program with specific inputs in a minimally instrumented environment, observing actual runtime behavior and failure signals [212]. Each execution follows a single concrete path determined by the provided input.
- *Dynamic Symbolic Execution (DSE)*: combines concrete execution with symbolic reasoning to explore alternative paths. By negating branch conditions and solving resulting constraints using SMT solvers, DSE generates new inputs that drive execution toward unexplored states [11]. While powerful, this approach incurs higher computational cost due to constraint solving and repeated path exploration [22].

A widely adopted form of DCE in security testing is *fuzz testing*. *Fuzzing* executes a target program with automatically generated inputs in order to expose crashes or violations of security policies [141, 149]. Fuzzers are commonly categorized by the degree of program insight they use:

- *Black-box fuzzers*: generate inputs without internal program knowledge, observing only input/output behavior.
- *Grey-box fuzzers*: employ lightweight instrumentation—typically coverage feedback—to guide input mutation toward unexplored paths.
- *White-box fuzzers*: incorporate full program semantics, often through symbolic reasoning, to systematically explore execution paths [149, 212].

Grey-box fuzzing has emerged as the dominant paradigm in large-scale vulnerability discovery because it balances scalability with execution feedback. By mutating inputs based on coverage signals, grey-box fuzzers efficiently explore diverse execution paths and generate minimized crash-inducing inputs. These crash artifacts form the empirical substrate for reproducible vulnerability instances and underpin continuous fuzzing infrastructures used in modern open-source ecosystems.

3.2.4 Fault Localization

Fault localization aims to guide debugging by identifying program locations responsible for faulty behavior. Prior work distinguishes two related activities: Fault Localization (FL) and *fix locus localization* [70]. Traditional FL seeks to identify the exact faulty statements that caused the failure, whereas fix locus localization identifies statements whose modification may repair the behavior, even if they are not themselves faulty. Because security vulnerabilities represent a subset of software faults—defined through weaknesses, errors, and failures [5]—techniques developed for general fault localization are directly relevant to vulnerability analysis. Comprehensive surveys classify FL techniques into several categories, among which slice-based and spectrum-based approaches are the most widely adopted [236].

Slice-based fault localization relies on program slicing, which extracts a reduced representation of a program that preserves behavior relevant to a variable or execution point [55, 9]. A slice contains statements that may influence a given computation, thereby narrowing the search space for debugging. Extensions such as program dicing compute the set difference between slices obtained from different executions. In vulnerability contexts, differential slicing has been used to contrast faulty and non-faulty executions discovered by fuzzing [206]. The resulting differential execution slice highlights statements that contribute to divergent behavior and can be combined with crash traces to localize vulnerability-relevant code regions.

Spectrum-based fault localization (SBFL) instead relies on execution statistics. SBFL techniques compute a *suspiciousness* score for each program element based on coverage information collected from passing and failing executions [236]. Similarity coefficient formulas correlate statement execution frequency with observed failures, ranking statements by their likelihood of being faulty. This approach requires an oracle, typically a test suite containing both successful and failing runs, to establish behavioral contrast. In vulnerability settings, SBFL has been combined with concolic execution to automate detection and localization tasks. Because SBFL depends on sufficient diversity between passing and failing executions, its effectiveness is closely tied to the quality and representativeness of the available input set.

3.3 Related Work

This section reviews prior work relevant to constructing evaluation-ready vulnerability benchmarks. Subsection 3.3.1 examines reproducibility, exploitability, and existing benchmark efforts, highlighting trade-offs between scale, realism, environmental stability, and causal grounding. Subsection 3.3.2 discusses fuzzing as a scalable mechanism for producing reproducible, version-anchored crash artifacts through infrastructures such as OSS-Fuzz and related systems. Subsection 3.3.3 reviews crash-based statistical debugging and spectrum-based localization techniques that aim to infer bug-relevant code regions from execution evidence. Finally, Subsection 3.3.4

synthesizes these threads to identify the remaining integration gap—namely, the absence of benchmarks that simultaneously preserve reproducibility, scalability, and defect-level causal semantics—and motivates the research question of this chapter.

3.3.1 Reproducibility, Exploitability, and Vulnerability Benchmarks

Prior studies have highlighted the fundamental difficulty of reproducing real-world vulnerabilities. Mu et al. [168] provide the first large-scale measurement of this challenge, showing that even well-documented issues frequently omit crucial information such as precise versions, environmental assumptions, and complete triggering inputs. Their results demonstrate that fewer than 55% of CVEs can be reproduced without substantial manual inference, and that missing Proof-of-Concept (POC) material, dependency drift, and ecosystem decay are among the strongest predictors of failure. Subsequent analyses of Proof-of-Concept exploits reinforce this pattern. Dang et al. [46] find that most CVEs lack any PoC, and that existing PoCs often omit 30% or more of the operational details required for deterministic reproduction. Recent automated approaches attempt to close this gap—for example, Ullah et al. [222] employ large language models to infer missing artifacts, rebuild environments, and construct exploits, reproducing roughly half of 841 CVEs published in 2024-2025—but even these systems emphasize that environmental reconstruction remains the dominant barrier.

Exploitability as empirical ground truth. Executable triggers serve as the empirical bridge between abstract vulnerability reports and concrete failure states. Many benchmark initiatives assume this perspective, treating exploitation as the canonical form of reproduction. In such settings, an input that deterministically triggers a vulnerability provides a verifiable ground truth against which both *static analysis* and *dynamic analysis* techniques can be evaluated. This principle underpins early bug benchmarks such as ManyBugs and IntroClass [86], where triggers validate behavioral correctness; later security-oriented benchmarks, including curated exploit sets for JavaScript [14], Java [24], and C/C++ [261], extend this idea by requiring sanitizer-verified crashes or security-relevant side effects as evidence of vulnerability manifestation. However, these projects also report the high human cost of constructing and validating such triggers—often requiring hours or days of manual setup, patch analysis, and *environment reconstruction* per entry—which explains why exploit-driven benchmarks tend to remain small, typically on the order of tens to a few hundred cases.

Benchmark scale versus realism trade-offs. Synthetic datasets such as Juliet [219] and Toyota ITC [211] achieve breadth but suffer from reduced realism because their vulnerabilities are manually injected rather than grounded in historical data. Debugging-oriented datasets offer realism but typically lack security semantics. Highly curated security benchmarks excel in fidelity but do not scale, precisely because constructing reproducible, exploitable, and patch-validated instances is resource-intensive. Across all of these efforts, persistent structural limitations recur: (i) environmental reproducibility is rarely guaranteed as build systems and dependencies evolve; (ii)

triggering inputs are often missing or non-deterministic; (iii) patches are not systematically validated against test cases; and (iv) ground-truth causal information—linking the observable failure to the underlying defect—is almost absent.

Continuous fuzzing as a scalable alternative. More recently, *continuous fuzzing infrastructures* have opened a new path toward scalable reproducibility. OSS-Fuzz [202] and OSV [211] provide historically stable snapshots of real-world projects, precise commit ranges, *sanitizers*, and minimized test cases, which collectively form an unusually rich substrate for reconstructing vulnerabilities at scale. The ARVO system [154] demonstrates the promise of this direction by automatically reproducing more than 5,000 memory vulnerabilities across 250+ C/C++ projects using OSS-Fuzz artifacts, resolving missing dependencies, and validating failures using sanitizer reports. ARVO thus shows that continuous fuzzing pipelines can produce reproducible instances at orders of magnitude larger scale than traditional benchmark-construction efforts. Yet even in this line of work, reproduction is primarily defined as crash manifestation, and the link back to the *root cause*—the defect-level origin emphasized in Chapter 2—remains largely unaddressed.

3.3.2 Fuzzing for Vulnerability Reproduction

Fuzzing has long been one of the most effective empirical methods for uncovering the concrete failure states that underlie software vulnerabilities. As noted earlier, reproducibility is central to any vulnerability benchmark [95]; in practice, this often collapses to the existence of a proof-of-vulnerability (PoV) input that deterministically triggers the faulty behavior. Since manually crafting such inputs requires deep knowledge of program semantics and input formats, fuzzing provides a practical, automated alternative: by repeatedly executing a program on generated, often malformed inputs, fuzzers induce program crashes [149, 122]. These crashes, typically manifested as fatal signals (e.g., segmentation faults, aborts), are treated as violations under the canonical security policy for fuzz testing, enabling the discovery of memory corruption and related security flaws at scale. Fuzzing's enduring popularity stems from its conceptual simplicity, high scalability, and low deployment cost [149, 95]. Decades of empirical studies confirm its effectiveness across large bodies of open-source and commercial software [95]. Evaluations of fuzzing techniques typically rely on: (i) a comparative baseline, (ii) a representative suite of target programs, (iii) a measurable outcome—often the number of distinct bugs or crashes triggered—and (iv) controlled configurations such as seed sets and time budgets [122]. These methodological requirements have motivated the development of realistic, security-oriented benchmarks, such as Magma [95], which assembles 118 historically rooted, developer-confirmed bugs alongside PoVs.

Triage pipelines and directed fuzzing for reproduction. A key reason fuzzing excels in vulnerability discovery is the maturity of its triage pipeline. Modern fuzzers integrate mechanisms for deduplication, prioritization, and test-case minimization [149], producing compact, representative PoVs that simplify downstream analysis. Deduplication groups crashes by underlying bug; prioritization ranks them by likely exploitability; and minimization yields the smallest input that

still triggers the failure. These features not only support practical vulnerability response [51] but also make fuzzing an attractive foundation for reproducible benchmarks: each PoV is a validated, minimized, and unique manifestation of a real defect. Within this landscape, grey-box fuzzing has emerged as the dominant paradigm for large-scale vulnerability discovery. By combining lightweight instrumentation with feedback-directed mutation, grey-box fuzzers such as AFL [250] and libFuzzer [144] achieve high throughput and broad coverage. Directed variants extend this capability by steering execution toward user-specified targets [31, 196], which has proven especially effective for *bug reproduction*. Hawkeye [31], for instance, dramatically reduces time-to-crash across diverse real-world programs and has led to the assignment of multiple CVEs. Target-oriented hybrid fuzzers [196] and environment-search fuzzers [157] further demonstrate the feasibility of reproducing hundreds of CVEs and uncovering dozens of new vulnerabilities in complex protocol and GUI software. Together, these advances position fuzzing not merely as a discovery technique, but as a systematic mechanism for reproducing confirmed vulnerabilities.

Continuous fuzzing infrastructures at scale. The step toward scalability came with the emergence of *continuous fuzzing infrastructures*. Prior to 2016, no service systematically fuzzed open-source software at scale. Google's OSS-Fuzz [202] established the first such platform: a distributed cluster that continuously builds, fuzzes, and verifies hundreds (now over a thousand) of real-world projects. OSS-Fuzz automatically collects minimized PoCs, sanitizer reports, and regression information for every confirmed bug. These artifacts are integrated into the OSV database [80] via the OSS-Fuzz-Vulns pipeline [81], which applies automated bisection to infer precise "introduced" and "fixed" commits for thousands of issues. As of 2025, OSS-Fuzz has contributed to the identification and remediation of more than $13,000$ vulnerabilities and $50,000$ bugs across a broad ecosystem of security-critical libraries [82]. Recent work demonstrates that the structured artifacts produced by OSS-Fuzz and OSV enable large-scale reproducibility. Most notably, ARVO [154] reconstructs build environments for vulnerable and fixed revisions, compiles historical snapshots, and validates PoCs against both versions to confirm the presence or absence of crashes. ARVO successfully reproduced more than $5,000$ memory vulnerabilities across over 250 projects—orders of magnitude larger than any prior benchmark—offering strong evidence that continuous fuzzing pipelines yield reproducible, historically grounded vulnerability instances at unprecedented scale.

3.3.3 Crash-Based Statistical Root-Cause Localization via Fuzzing

Once a fuzzer yields crashing inputs, developers must still determine why a fault occurs and how to fix it. This analysis typically involves inspecting program state at the crash point, reasoning backward through the executed path, and identifying the instructions or data dependencies responsible for the failure [15]. Such retrospective debugging is difficult because most executed code behaves correctly, memory-related defects often manifest far from their origin, and crashes provide only a late symptom of deeper state corruption [15, 107]. As a result, root cause identification remains

one of the most time-consuming components of bug remediation, with maintenance studies reporting that developers spend a disproportionately large share of their effort localizing faults relative to assessing severity or implementing patches [131, 183, 107, 53]. The problem is magnified in fuzzing campaigns, where many PoCs correspond to the same underlying defect, forcing analysts to revisit duplicate failures [15, 131]. In practice, this manual symptom-to-cause reconstruction aligns closely with traditional debugging workflows.

Statistical debugging and spectrum-based localization. Research on automating this diagnostic step spans statistical debugging, Spectrum-Based Fault Localization (SBFL), and earlier state-differencing methods such as delta debugging. Although the literature often conflates these techniques with the broader goal of RCA, statistical debugging is best viewed as a framework that collects behavioral data across passing and failing executions, applies probabilistic or deterministic models, and ranks program elements by their association with failures [216, 252, 183, 207]. Within this framework, SBFL compares coverage spectra to identify statements whose execution correlates with failure or exploitability [207, 131]. Historically applied to general software failures, SBFL has been adapted for crash-centered vulnerability analysis, where spectrum-based ranking guides analysis toward fault-relevant regions in crashing binaries [53, 107, 131]. However, these approaches require sufficient variation across both crashing and non-crashing runs; without enough behavioral diversity, statistical signals weaken and localization accuracy degrades [207, 53].

Execution contrast as a limiting factor. In crash-oriented settings, the quality of available executions becomes a central limitation, as traditional SBFL assumes a well-structured test suite that separates passing from failing behavior—an assumption that rarely holds for real-world vulnerabilities, where analysts often possess only a binary and a single PoC [53, 183]. Although crash-based variants of statistical debugging treat crashing and non-crashing runs as the necessary classes, naïvely generated inputs often fail to meaningfully diversify control flow, leading to indistinguishable spectra, inflated *suspicion scores* for high-frequency but irrelevant statements, and signals that overfit the original crash path [207, 53]. These issues worsen when the test distribution is unbalanced or when there is no basis for selecting the input population used to estimate fault–failure correlations. Consequently, in the absence of curated tests and sufficient behavioral contrast, statistical and spectrum-based techniques alone frequently yield low-fidelity localizations.

Fuzzing-guided crash-based localization. To address these constraints, crash-based statistical debugging integrates spectrum-based reasoning with targeted input generation—most commonly via fuzzing or directed mutation [183]. Starting from a single crashing input, these approaches synthesize additional executions that remain near the original crash while exploring alternative paths and data values, thereby producing the behavioral variation necessary for statistical inference [15, 207, 53, 183]. Systems such as AURORA [15], the concentrated test-suite approach of

Shen et al. [207], the refinement strategy of Du et al. [53], and BENZENE [183] demonstrate that guided input generation can substantially improve localization accuracy across real-world CVEs. While empirical results vary by technique and benchmark, these studies collectively indicate that combining fuzzing with statistical debugging can alleviate test-quality limitations by generating executions that are sufficiently diverse to reveal discriminative signals and constrained enough to preserve causal relevance—thereby improving the prospects of reliable crash-based localization.

3.3.4 Summary and Research Gap

Prior work reveals complementary advances but also persistent structural gaps. Existing vulnerability benchmarks provide empirical foundations for evaluation, yet they face trade-offs among scale, realism, and root cause localization. Synthetic datasets scale but lack realism; curated real-world collections preserve realism but remain limited in size and reproducibility; and most benchmarks characterize vulnerabilities at the level of their manifestation rather than at the level of the underlying bug. Consequently, they rarely preserve the link between the observable failure and the bug that causes it.

Fuzzing infrastructures address part of this limitation by supplying reproducible crash artifacts tied to vulnerable versions at scale. Continuous pipelines maintain triggering inputs, sanitizer evidence, and commit-bounded changes, thereby strengthening the manifestation side of the ground truth. Complementary research in crash-based localization and statistical debugging advances techniques for identifying bug and fault locations from execution context, but these techniques focus on the debugging workflow rather than on the construction of evaluation-ready datasets. Consequently, fuzzing defines ground truth at the level of manifested crashes, localization research operates outside benchmark construction, and existing benchmarks seldom record both the manifestation and the bug within the same structured instance.

What is missing is a benchmark that preserves reproducible crash artifacts and the distinction between where a vulnerability manifests and where its bug originates. Without such artifacts, it is unclear what correct detection should mean in evaluation settings. This motivates the central question of this chapter (RQ2): **which vulnerability attributes must a benchmark preserve to support reliable evaluation?** Answering this question requires not only identifying those attributes in principle, but constructing benchmark instances that concretely retain them—thereby aligning reproducible crash artifacts with root cause localization.

3.4 Methodology

This section describes the pipeline used to construct the benchmark from OSS-Fuzz disclosures. First, reproducible OSS-Fuzz vulnerabilities are extracted from OSV and filtered to retain only entries with resolved issue reports, structured crash reports, and retrievable introducing and fixing revisions (Subsection 3.4.1). These entries are then linked to CVEs in the NVD through product alignment, CWE overlap, reference evidence, and disambiguation (Subsection 3.4.2). Each linked vulnerability is then rebuilt at the crash-inducing revision, its original test case is replayed to

reproduce the crash, and the fixing revision is validated to confirm that the failure no longer occurs (Subsection 3.4.3). Finally, the bug location is identified by aligning stack trace locations with the diff between the crash and fixing revisions, the separation between bug and crash locations is computed from the reproduced stack traces, and both locations are labeled using the labeling schema defined in Chapter 2 (Subsection 3.4.4; Section 2.4.5).

3.4.1 Extracting vulnerability reproduction metadata for OSS-Fuzz from OSV

Constructing the benchmark begins with a population of vulnerabilities that are not only real, but also supported by the historical, behavioral, and trace artifacts required for reconstruction and validation. OSS-Fuzz vulnerabilities curated in OSV provide such a starting point because they originate from systematic fuzzing, are accompanied by structured crash reports, and are tied to version history through automated revision analysis. However, OSV entries vary in the completeness of the available artifacts. This step therefore serves as a qualification phase that retains only records containing the metadata required in later stages. Rather than collecting this information incrementally, we enforce these conditions at extraction time so that subsequent stages operate on entries already suitable for reconstruction.

The qualification process proceeds through four staged checks, each enforcing a distinct requirement. First, we validate entry status, retaining only confirmed, non-withdrawn vulnerabilities to exclude superseded or invalidated reports. Second, we require resolution of an OSS-Fuzz issue tracker reference, ensuring that each entry is linked to its underlying fuzzing report and the operational context required for reproduction (Appendix B.1.1). Third, we extract structured crash reports and map crash types to CWE identifiers, providing a normalized description of how the vulnerability is reported to manifest at runtime (Appendix B.1.2). Fourth, we verify historical range availability by requiring retrievable introducing and fixing revisions from a resolvable repository, thereby bounding the bug in version history (Appendix B.1.3). Together, these stages ensure that each retained vulnerability is valid, reproducible in context, characterized on the manifestation side, and linked to recoverable development history.

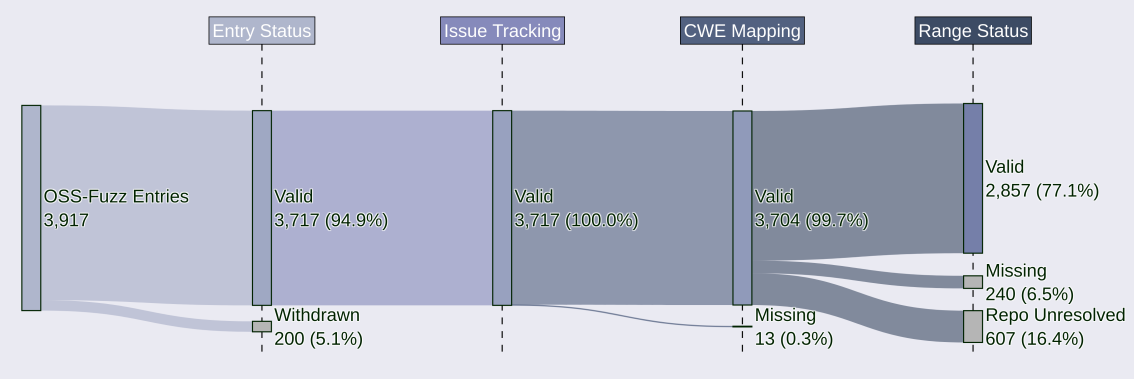


Figure 3.1: Sankey diagram summarizing the filtering of OSS-Fuzz OSV entries by Entry Status, Issue Tracking, CWE Mapping, and Range Status.

Figure 3.1 illustrates this staged filtering as a progressive reduction from the full set of 3,917 OSS-Fuzz-derived OSV entries to the subset that satisfies all requirements. After entry status validation, 3,717 records (94.9%) remain, with 200 entries (5.1%) excluded due to withdrawal. Issue-tracker traceability is available for all remaining entries, preserving the full set at this stage. Crash report qualification then removes 13 cases (0.3%) lacking sufficiently structured crash descriptors, leaving 3,704 entries (99.7% of the previous stage). The final historical range check yields 2,846 entries (76.8% of the original population) whose introducing and fixing revisions are retrievable, while 253 cases (6.8%) lack complete range information and 605 cases (16.3%) reference repositories or revisions that are no longer resolvable.

3.4.2 Linking OSS-Fuzz Vulnerabilities with CVEs in the NVD

The vulnerabilities qualified in the previous step provide crash reports and version history, but they remain tied to the fuzzing ecosystem rather than to the CVE records used in disclosure and reporting. Linking these entries to CVEs is therefore necessary to connect two complementary views of the same vulnerability: OSS-Fuzz provides reproduced crash evidence and development history, while CVEs place vulnerabilities within widely used reporting practices and security classifications. Establishing reliable correspondences allows each benchmark instance to carry both reproduced crash evidence and standardized vulnerability descriptors. Because individual metadata fields are incomplete or noisy, this linkage cannot rely on any single signal; instead, it requires a structured combination of multiple independent sources of evidence.

We implement a staged matching workflow in which each stage acts as a filter that progressively increases confidence in OSS-Fuzz to CVE correspondence. The first stage enforces *product alignment*, where OSS-Fuzz repositories are translated into the vendor-product identifiers used in CVE configurations, and only pairs with overlapping products are considered (Appendix B.2.1). The second stage performs *CWE overlap* by requiring intersection between the CWE(s) derived from OSS-Fuzz crash semantics and the CWE classifications listed for the CVE, ensuring compatibility in weakness type (Appendix B.2.2). Together, these stages establish that candidate pairs refer to the same software system and a compatible class of weakness. The third stage evaluates *reference evidence*, checking whether CVE reference links resolve to OSS-Fuzz issue reports or commits that coincide with the introduced or fixed revisions recorded for the OSS-Fuzz entry; such overlaps provide direct historical linkage signals (Appendix B.2.3). When multiple candidates remain, a final *disambiguation* step compares program entity tokens in CVE descriptions (e.g., function or file names) with identifiers observed in OSS-Fuzz crash stack frames to resolve residual ambiguity, after which a priority-based rule retains only the strongest supported correspondence (Appendix B.2.4). Only matches supported by a coherent combination of these signals are carried forward.

Figure 3.2 summarizes the effect of this workflow on the 2,846 metadata-qualified OSS-Fuzz entries carried forward from the previous step. Product alignment is possible for 2,395 entries (84.2%), while 451 entries (15.8%) lack a compatible product mapping and are excluded from further consideration. Among product-aligned entries, 1,520 (63.5%) exhibit CWE overlap with

at least one CVE, whereas 875 (36.5%) do not meet the weakness-alignment criterion. Reference evidence is present for 76 of the CWE-aligned candidates (5.0%), representing the subset where explicit provenance signals connect the two sources. After disambiguation and consolidation, these correspondences yield 76 CVE matches, of which 70 (92.1%) map uniquely to a single OSS-Fuzz entry and 6 (7.9%) involve multiple plausible candidates before final resolution. This yield is consistent with prior observations on OSS-Fuzz to CVE linkage. Keller et al. [115] report that 1,731 out of 18,794 fuzzing issues (9.2%) could be matched to a CVE using project alignment and relaxed CWE matching based on ancestry. In contrast, our workflow enforces stricter criteria, requiring product to project alignment, direct CWE correspondence, and reference evidence, which results in a lower but higher confidence linkage rate. The quality of the resulting correspondences is assessed through manual validation in Section 3.5.1, yielding a precision of approximately 97.1%.

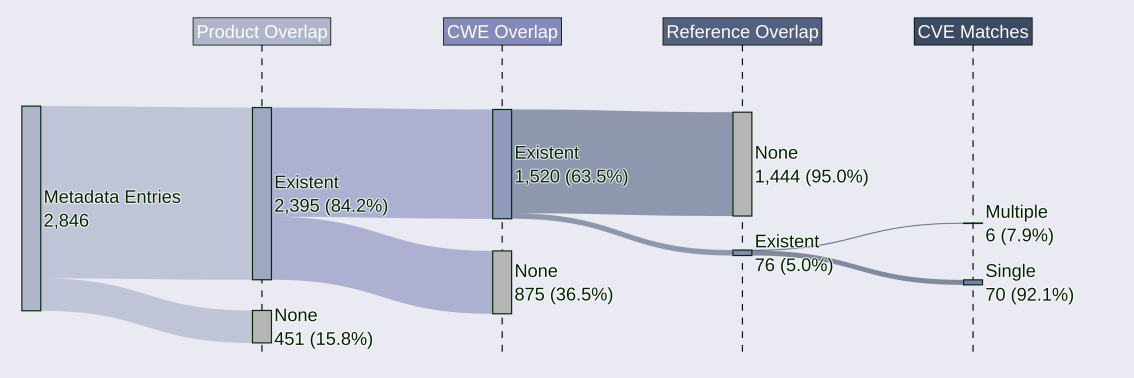


Figure 3.2: Sankey diagram illustrating the OSS-Fuzz to CVE linkage workflow, filtering by product overlap, CWE overlap, and reference evidence leading to final CVE correspondences.

3.4.3 Reproducing and Verifying OSS-Fuzz CVEs Automatically

A defining requirement of our benchmark is that every retained vulnerability must be reproducible, supported by the vulnerable version, and empirically validated. The objective of this stage is therefore to ensure that OSS-Fuzz vulnerabilities linked to CVEs are not merely metadata correspondences, but concrete, buildable, and verifiable instances of real bugs. This requirement is operationalized through two complementary execution modes: (i) crash reproduction in the introducing revision identified through OSS-Fuzz metadata, and (ii) fix verification in the revision containing the OSV-designated fixing commit. Only vulnerabilities that satisfy both conditions are retained; any case that cannot be reconstructed, triggered, and validated against its recorded fix is excluded. The operational architecture implementing these workflows—including execution modes, orchestration stages, and structured status reporting—is detailed in Appendix B.3.1.

Conceptually, reproduction proceeds by resolving the OSV record to its associated OSS-Fuzz issue report and extracting the artifacts required to reconstruct the execution context. This includes retrieval and normalization of crash metadata and determination of the appropriate build snapshot

(Appendix B.3.2). The project's source state and configuration are then reconstructed from the snapshot-defined repository revisions and OSS-Fuzz project descriptors (Appendix B.3.2). Compilation and execution are performed inside containers configured to mirror OSS-Fuzz's build and runtime infrastructure, ensuring consistent instrumentation, sanitizer configuration, and architecture (Appendix B.3.3). The original crashing test case is executed under this reconstructed environment, and reproduction is considered successful only if the observed sanitizer output matches the structured crash report recorded in the issue report. Crash parsing and equivalence checking—including crash type, operation, and stack trace locations—are formalized in Appendix B.3.4. Fix verification mirrors this workflow at the fixing revision and requires that the crash is absent there. Only vulnerabilities that satisfy both crash matching in the introducing revision and absence of the crash in the fixed revision are retained in the benchmark.

3.4.4 Localizing Vulnerability Root Causes from Crash and Patch Artifacts

After crash reproduction and fix verification, the next step is to identify the bug location in source code. The benchmark requires each vulnerability to be tied to a concrete bug location, not only to the line where the crash occurs. Bug localization is therefore defined as a procedure that combines stack trace information with the changes introduced by the fixing commit. The search space is bounded by two commit versions: the crash-inducing revision and the fixing revision obtained during reproduction. From these, stack trace locations and the corresponding file diff in the patch are extracted (Appendix B.4.1). Candidate locations are ranked using a line-distance heuristic and then manually reviewed before final selection (Appendix B.4.2).

The procedure uses three concrete artifacts. First, the crash-inducing and fixing commits define the before and after code states. Second, structured sanitizer output provides stack trace locations indicating where the vulnerability manifested (Appendix B.4.1). Third, the fixing commit provides the edited lines. For each crash-implicated file, the diff between the two commits is computed and restricted to those files. Modified lines are compared to crash line numbers using absolute line distance (Appendix B.4.2). The candidate with the smallest distance is selected, with deterministic tie-breaking when needed. The selected location is then manually validated against the fixing commit and surrounding context to confirm that it represents the bug location.

The structural relation between the bug location and the crash location is measured using call-graph separation. Each reproduced stack trace is treated as an observed execution path. Separation is defined as the number of caller-callee edges between the manually validated bug function and the crash function extracted from sanitizer output (Appendix B.4.3). When both functions appear in available traces, separation is computed using a lowest common ancestor formulation over the stack paths; when stack information is incomplete, the default rule is applied (Appendix B.4.3). After localization, both locations are labeled using the labeling schema defined in Chapter 2 (Section 2.4.5). Each location is assigned both a CWE and a BF operation.

3.5 Results

Building on the preceding methodology, this section examines the benchmark instances retained after linkage, validation, and reproduction. Section 3.5.1 first summarizes the resulting benchmark scope after linkage and manual validation, and Section 3.5.2 then reports end-to-end reproduction yield, highlighting the most frequent build failures that limit reproduction. From the reproduced cases, Section 3.5.3 measures the separation between bug and error locations, and Section 3.5.4 assesses and compares the operations at bug and error locations, along with their assigned CWEs. Finally, Section 3.5.5 jointly examines separation and assessment to answer the research question.

3.5.1 Benchmark overview

This benchmark is derived from OSS-Fuzz vulnerabilities published in OSV and linked to CVE disclosures, focusing on C/C++ projects with issue activity roughly spanning 2017 to 2025. In total, the dataset covers 41 distinct projects and is constructed to support later evaluation using version-specific artifacts such as commits, crash reports, and reproducer inputs.

The OSV to CVE linkage is produced automatically by the pipeline described in Section 3.4.2. The automation yields 70 candidate OSV–CVE mappings. These candidates are then manually validated by checking available signals across CVE-ID, OSV-ID, and OSS-Fuzz issue IDs, including crash file and function identifiers and operation names. After validation, 69 CVEs are retained: one entry is removed as a duplicate, and one is corrected after identifying a wrong match caused by a very similar OSV entry. For example, CVE-2021-36088 was initially linked to the wrong OSV entry; the mapping was corrected by checking affected versions and selecting the correct OSV-2021-702 match for CVE-2021-36088. This corresponds to a precision of approximately 97.1% (68/70) for the automated linkage. Recall is not estimated because a complete set of true matches in the OSV database is not available.[1]

From these 69 linked CVEs, historical reconstruction and reproduction identify which cases can be converted into executable benchmark instances. Overall, 40 cases were successfully built under reconstruction, and 30 were reproduced with crash matching the corresponding OSS-Fuzz crash report, forming the executable core used throughout the remainder of this chapter. The remaining linked cases fail to build or do not yield a matching crash during replay, due to practical reconstruction limits analyzed in Section 3.5.2.

For reproduced entries, two complementary artifacts are retained for subsequent analyses: (i) crash reports that localize where and how the vulnerability manifests at runtime, including crash location, crash type, and sanitizer context; and (ii) the fixing patch, which identifies the bug location. Quantitative breakdowns of manifestation labels and root cause labels are deferred to Section 3.5.4, where the difference in operation diversity between manifestation and bug is analyzed explicitly.

[1] To compute recall properly, all true matches that exist in the OSV database would be needed.

3.5.2 Reproduction outcomes

This subsection reports the manually validated outcomes of replaying OSS-Fuzz test cases on vulnerable revisions, focusing on whether crashes occur and how closely they align with the original reports. Reproduction outcomes are categorized into build failures, successful builds matching crashes, successful builds mismatching crashes, and successful builds without crashes; the exact crash matching criteria are defined in Appendix B.3.4. Table 3.1a summarizes these outcomes, and Table 3.1b compares CWE distributions in the reproduced subset with the full set.

Table 3.1: Reproduction outcomes and representativeness of reproduced cases.

(a) Reproduction outcomes and failure categories

Stage / outcome	Count	Share
Linked CVEs	69	100%
Builds	40	58.0%
Crashes	32	46.4%
Matches	30	43.5%
Mismatch	2	2.9%
No crash	8	11.6%
Build fails	29	42.0%
Build errors	14	20.3%
Dependency issues	13	18.8%
Repository issues	2	2.9%

(b) Crash-level CWE distribution comparison

CWE-ID	All (%)	Reprod. (%)
CWE-125	14.49	16.67
CWE-415	5.80	6.67
CWE-416	8.70	13.33
CWE-476	2.90	3.33
CWE-787	60.87	60.00
Others	7.25	0.00

Others: CWE-121, 122, 672, 908

Among the 69 linked entries, 40 build successfully. Of these buildable cases, 32 crash under replay and 8 do not; of the crashing cases, 30 match the OSS-Fuzz report and 2 do not. For illustration, `OSV-2020-2132` reproduces as a `READ` of 4 bytes in `flb_hash_get_by_id`, while the report indicates a `WRITE` of 1 byte in `flb_gzip_compress`; `OSV-2020-2305` reproduces as a `SEGV` on a `READ` with no location specified in the ASan logs, while the report indicates an `UNKNOWN WRITE` in `tb_flush_armeb`. The CWE distribution of the 30 matched crashes closely follows that of the full set: CWE-787 (out-of-bounds write) remains dominant in both (60.87% vs. 60.00%), with comparable shares for CWE-125, CWE-415, and CWE-476, and a small tail of other cases (7.25%) not represented among reproduced entries.

The remaining 29/69 entries fail to build and therefore do not yield executable instances. Build failures fall into three recurring categories: repository issues, dependency issues, and build errors. Repository issues include automated checkout blocked by access or hosting constraints (e.g., `OSV-2025-74`; `OSV-2020-1386`). Dependency issues include missing harness repositories or unavailable build artifacts, as the missing fuzzing dependencies for wolfMQTT entries [2], and missing build-time artifacts in `OSV-2021-1627` and `OSV-2020-254`. Build errors include toolchain or build-system drift, such as the Autoconf requirement in `OSV-2025-131`, modern compiler diagnostics treated as errors in `OSV-2020-680` and `OSV-2021-775`, and linker-stage failures such as `OSV-2021-557` and `OSV-2020-744`.

[2] OSV-2021-1204, OSV-2021-1211, OSV-2021-1348, OSV-2021-1349, OSV-2021-1353, and OSV-2021-1361

Finding 1. The reproduced subset of vulnerabilities is broadly representative of the full set in terms of CWE distribution. Among 69 linked entries, 40 build successfully and 30 produce crashes that match the original OSS-Fuzz reports, with CWE-787 remaining the most frequent in both sets and similar shares for other common CWEs. This makes the 30 matched crashes a reasonable basis for subsequent analyses in this chapter.

3.5.3 Function-level and line-level separation between bug and error locations

The 30 reproduced vulnerabilities provide manually validated bug and error locations, allowing assessment of how far the locations where the crash is observed may be displaced from the location where the bug is introduced or fixed. When the two locations reside in different functions, separation is measured using call-graph distance, defined as the number of call edges between the bug function and the error function. When both reside in the same function, separation is measured using within-function line distance $\Delta = |line_{\text{bug}} - line_{\text{error}}|$. Full definitions and separation categories are given in Section 3.4.4 and Appendix B.4.3.

Figure 3.3 summarizes bug and error separation at the function level. Thirteen of the 30 vulnerabilities cross function boundaries: eight span different files, while five remain within the same file but occur in different functions. Bug–error pairs in different files range from one or two call edges up to nine edges, while same-file but different-function pairs span one to four edges.

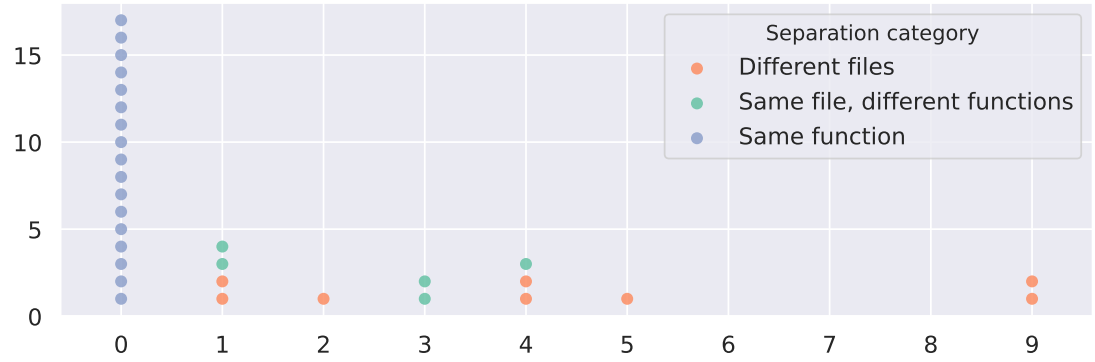


Figure 3.3: Bug and error separation by call-graph distance (x-axis). Each dot represents one reproduced vulnerability; the y-axis encodes counts.

Among the 17 cases in the same function (call-graph distance 0), Figure 3.4 shows the within-function line separation. Bug and error locations coincide exactly ($\Delta = 0$) in six cases, while the remaining cases show non-zero separation ranging from small single-digit distances ($\Delta \in \{3,5,6,7\}$) to substantially larger gaps, with a maximum observed separation of 155 lines.

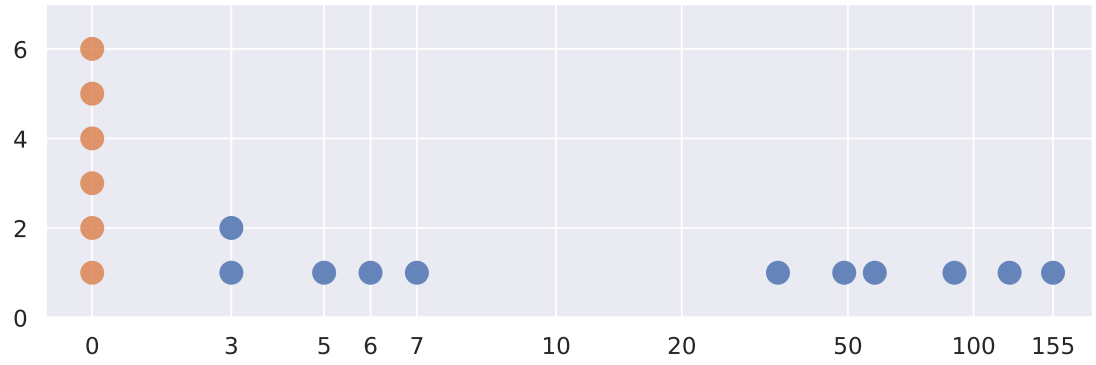


Figure 3.4: Within-function bug and error line separation (x-axis on a symmetric log scale to show both small and large separations); each dot represents one reproduced vulnerability; the y-axis encodes counts.

Finding 2. In the reproduced vulnerabilities, bug and error locations frequently do not coincide. Thirteen of the 30 cases cross function boundaries, with call-graph distances reaching up to nine edges. Among the remaining 17 cases in the same function, bug and error locations coincide in only six; the others show line separations up to 155 lines.

3.5.4 Assessment of the operation at the bug and error locations

This subsection characterizes the operation at the error location and the root cause at the bug location for the 30 reproduced benchmark instances. Error assessment examines the weakness type used during OSV–CVE linkage, the error CWE, and the concrete memory operation reported at runtime by the sanitizer, while bug assessment examines the cause and operation that define the root cause, along with their associated CWE. Figure 3.5 summarizes error assessment, and Figure 3.6 summarizes bug assessment.

Error assessment. Among the 30 reproduced vulnerabilities, 19 are linked to NVD records via a *secondary* CWE derived from the error operation, while the remaining 11 are matched via a *primary* CWE shared between the OSV and NVD records. For the error CWE, entries are concentrated in a small set of manifestation categories: CWE-787 (out-of-bounds write, 18 cases) and CWE-125 (out-of-bounds read, 5 cases), with smaller numbers of CWE-416 (4), CWE-415 (2), and CWE-476 (1). Most sanitizer reports attribute errors to either a WRITE (19 cases) or READ (9 cases) memory access. Two reproduced cases lack an explicit operation label; both correspond to CWE-415 (double free), where sanitizer reports describe an "attempting double-free" without attributing the error to a specific operation.

Bug assessment. Among the 30 reproduced vulnerabilities, bugs are dominated by *missing* cause (20/30, 66.7%), with the remainder classified as *incorrect* cause (10/30, 33.3%). These

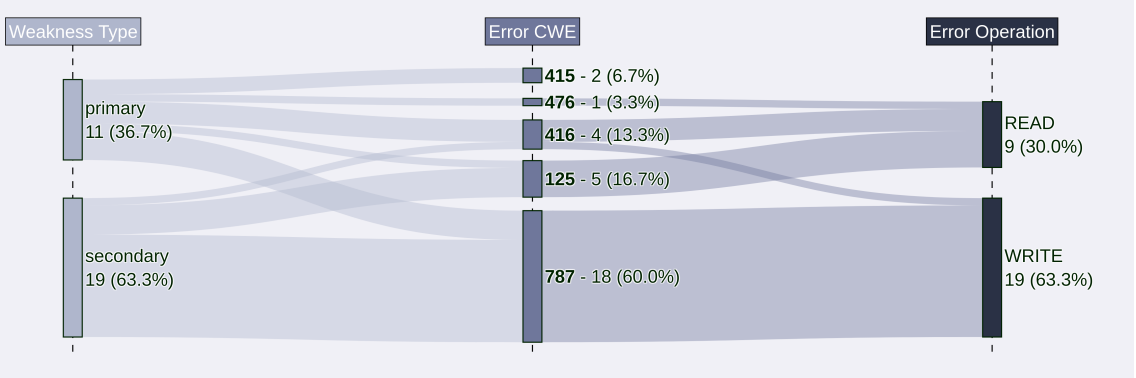


Figure 3.5: Sankey flow for error assessment: weakness type to error CWE to error operation.

causes primarily manifest as missing or incorrect VERIFY operations (21, 70.0%), while the remaining cases involve CALCULATE (2), CLEAR (2), and single instances of EVALUATE, REPOSITION, REASSIGN, INITIALIZE, and CALL. These eight distinct operations map to a diverse set of CWEs, with *CWE-1284* (6), *CWE-129* (5), *CWE-130* (3), *CWE-754* (3), and *CWE-131* (2) occurring most frequently, and eleven CWEs that appear only once. In some cases, the CWE assignment only approximates the bug because the taxonomy conflates the cause with the consequence.[3] The CWE set for the bug does not overlap with the CWE set for the error.

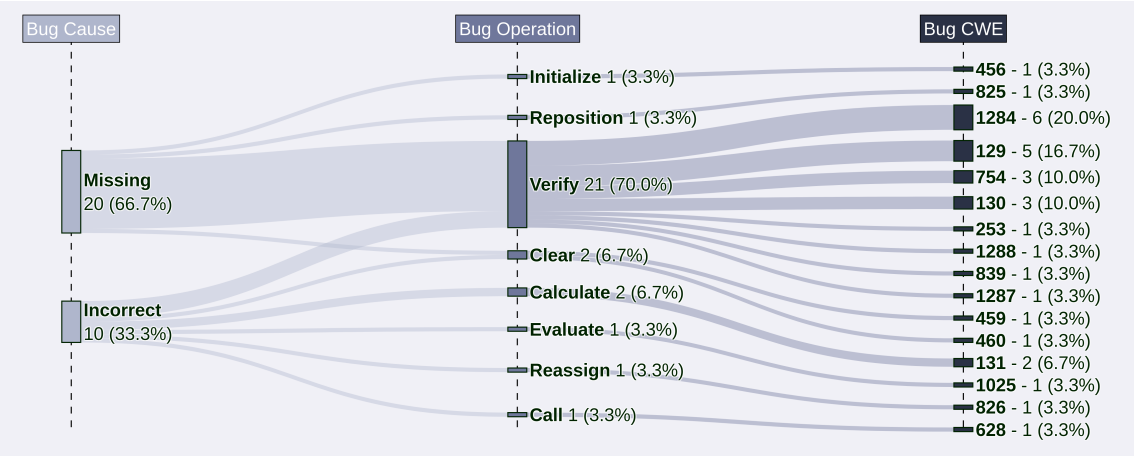


Figure 3.6: Sankey flow of bug assessment: cause to operation to CWE.

Finding 3. Error locations cover only two memory operations, while bug locations span eight distinct operations. Most error operations are WRITE (19 cases) and map to CWE-787, appearing as secondary CWEs in the NVD metadata. At the bug location, most operations are VERIFY (21 cases) under a *Missing* cause (20 cases), aligned with a broad set of CWEs.

[3] For example, an invalidated-pointer dereference after resizing is naturally captured using a dereference CWE (e.g., *CWE-825*), even when the bug is a missing pointer re-acquisition.

3.5.5 Cross-reference analysis of separation and assessment

This subsection examines bugs and errors across the 30 reproduced benchmark instances by cross-referencing separation and assessment. Bug–error separation categories, defined in Appendix B.4.3, span cross-file, cross-function, and same-function cases, distinguished by $\Delta > 0$ and $\Delta = 0$; assessment spans bug cause, bug operation, aggregated by the categories in Appendix A.2.1, and error operation. Figure 3.7 summarizes the intersection of these attributes.

Missing bugs manifest more frequently within the same function (13 of 20 cases across the two same-function rows), while *Incorrect* bugs manifest more frequently outside the same function (6 of 10 cases across the cross-file and cross-function rows), with no cases coinciding with the error. DATA CHECK bugs manifest more frequently within the same function (15 of 20 cases across the two same-function rows), while DATA TYPE and MEMORY bugs manifest more frequently outside the same function (7 of 9 cases across the cross-file and cross-function rows). There are no DATA TYPE bugs coinciding with the error, and no MEMORY bugs manifesting within the same function with $\Delta > 0$. At the error location, READ occurs more frequently within the same function (7 of 9 cases across the two same-function rows), while WRITE shows a more varied distribution across separation categories, though it still concentrates within the same function with $\Delta > 0$.

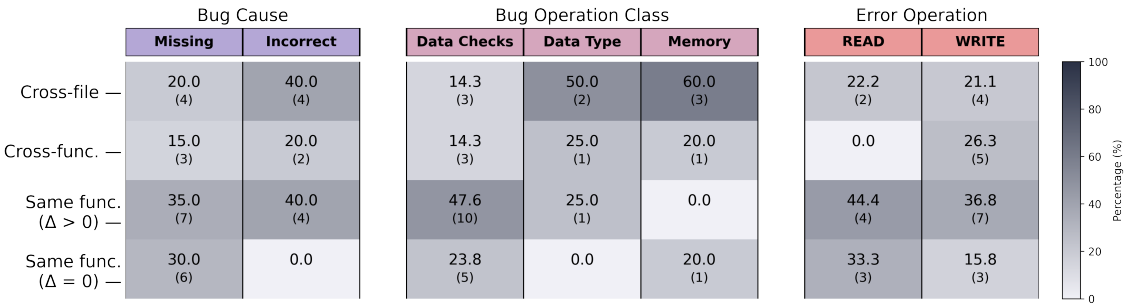


Figure 3.7: Heatmap cross-referencing bug–error separation (rows) with assessment (columns).

Finding 4. Bug–error separation splits assessment into same-function and outside-same-function cases. Within the same function, assessment clusters on *Missing* DATA CHECK bugs, while outside it, bugs shift toward *Incorrect* DATA TYPE and MEMORY. At the error location, READ clusters within the same function, while WRITE spreads across categories. These results show bug-error separation and assessment as relevant attributes for evaluation.

3.6 Discussion

This discussion synthesizes the findings from Sections 3.5.2–3.5.5 and situates them in the broader challenge of designing benchmarks that support reliable evaluations. Section 3.6.1 interprets what the reproduced vulnerabilities reveal about bug–error separation and assessment, clarifying how the structural and semantic attributes of the benchmark relate. Section 3.6.2 examines the implications for empirical researchers, tool builders, and practitioners, arguing that these must handle the

full vulnerability chain because its structure imposes distinct localization and assessment requirements. Section 3.6.3 addresses the conditions under which these findings hold, covering selection bias, labeling ambiguity, and infrastructure fragility, and how each constrains external validity, construct validity, and corpus stability. Section 3.6.4 identifies directions that would extend the pipeline's scope, strengthen label reliability, and improve scalability beyond the current corpus.

3.6.1 Interpretation of results

This subsection interprets the results from Sections 3.5.2–3.5.5 in terms of what the reproduced vulnerabilities reveal about the attributes needed for evaluation. The interpretation is scoped to the 30 reproduced cases and the taxonomies used to assess them. It first characterizes the reproducible subset and the conditions under which it was obtained, then examines bug–error separation as a structural attribute, followed by bug–error assessment as a semantic attribute, and concludes with the interdependence of these attributes for evaluation.

Representativeness of the reproducible subset. The reproduction procedure recovers 30 of 69 linked entries as validated, crash-matching instances, while build errors and missing dependencies dominate the remaining cases. These unreproduced cases mostly reflect ecosystem drift and practical engineering constraints, such as repository availability, dependency breakage, and toolchain changes, rather than any clearly observable pattern in vulnerability type. Among the 40 entries that build, the 30 reproduced crashes closely follow the CWE distribution of the full set, with no clear over- or under-representation of major vulnerability classes. The 30 reproduced cases therefore provide a reasonable empirical basis for the analyses in this chapter, while recognizing that the subset is conditioned on vulnerabilities that remain buildable under current ecosystem conditions.

Bug–error separation as a structural property of the vulnerability. The results show that bug and error locations do not reliably coincide, and that this divergence operates at two distinct levels: across function boundaries in the majority of cases, and across non-trivial line distances even within the same function. This supports the vulnerability model introduced in Chapter 2, in which bugs propagate through intermediate errors before manifesting as failures; the observed separation is exactly what that propagation chain predicts. Within this benchmark, the error location reflects only the endpoint of that chain, not its origin. Even in the six cases where both locations coincide, the code at the error location describes what is observable rather than what causes it; a distinction that the assessment attribute examines.

Distinct improper operations converge into memory errors at the state level. The assessment results show that bugs and errors reflect structurally different aspects of the same vulnerability: the bug is an improper operation in the code, while the error is an improper state in memory. At the error location, two memory-usage operations, READ and WRITE, propagate the improper states. At the bug location, operations span eight distinct types, predominantly VERIFY under the *Missing* cause, covering a broader and entirely disjoint set of CWEs. This shows that many

distinct improper operations at the code level converge into a small set of memory errors at the state level; a convergence that reflects the manual memory management of C/C++ programs, where improper states propagate until they surface as memory failures. Within this benchmark, the CWE assignment at the bug location approximates the cause, since the taxonomy does not consistently distinguish cause from consequence.

Separation and assessment as interdependent attributes. The cross-reference between separation and assessment shows that same-function cases cluster on *Missing* DATA CHECK bugs, while outside-same-function cases shift toward *Incorrect* DATA TYPE and MEMORY. This means the nature of the bug reflects how far it lies from where it manifests. Bug operations broaden as separation increases, while error operations remain concentrated across all separation categories. This asymmetry shows that error operations alone do not carry sufficient information to distinguish one vulnerability from another. Within this benchmark, the pattern suggests that both separation and assessment are relevant attributes for evaluation: retaining only the manifestation conflates distinct bugs, and risks reducing evaluation to incidental agreement with error locations.

3.6.2 Implications for research, tooling, and practice

The results presented in this chapter carry implications for empirical researchers, tool builders, and practitioners and maintainers working with vulnerability benchmarks. These implications are grounded in the interpretation of the benchmark findings, and speak to how vulnerabilities should be measured and assessed, what capabilities tools operating on them must support, and what conditions must hold for such benchmarks to be assembled and sustained.

For empirical research. Because bug and error locations do not always coincide, empirical measurements of tools on these benchmarks must distinguish between whether a tool matches the bug location or the error location. A tool that localizes the error location may capture the manifestation of a vulnerability while missing the bug location; conversely, a tool that localizes the bug location may not match the error location. Measuring only one of these locations can therefore bias conclusions about tool effectiveness. This distinction is particularly relevant to empirical evaluations of static analysis tools, which often report locations that match the manifestation of a vulnerability rather than its root cause. In such settings, evaluations limited to error locations may overstate the detection and assessment capabilities of tools.

For automation and tooling. Bug–error separation and assessment together show that tools must handle the full vulnerability chain, not just the final manifestation. Because bug and error locations range from coincident to cross-file separation, tools must not only detect locations along the vulnerability chain but also disambiguate what is being reported at each location. Because bugs span diverse operations while error operations concentrate on memory usage, the two stages of the chain impose distinct assessment requirements. In practice, this means that tooling should

support both bug and error localization and classify the operations that characterize each stage of the chain, with bug localization being the least addressed of the two.

For maintainers and practitioners. Because build errors account for most unreproduced cases due to repository unavailability, dependency breakage, and toolchain changes, crafting a benchmark depends on what project maintainers provide and sustain. To reproduce a vulnerability, the project must remain available, its dependencies obtainable, and its build process able to compile the program needed to rerun the crashing input. Beyond infrastructure, maintainers also determine the benchmark metadata: because root-cause labels are derived from patches, entries are limited to crash types. If maintainers reported bug locations and root-cause labels as part of their patching workflow, the pipeline could produce more complete and classifiable benchmark entries with less manual intervention. For practitioners, any benchmark that depends on open-source infrastructure should be treated as a snapshot rather than a stable ground truth, since coverage reflects what the ecosystem supported at the time of assembly, and keeping it current requires active maintenance.

3.6.3 Limitations and threats to validity

This study has several limitations that should be considered when interpreting the results. These concern the selection and sample size of the reproduced corpus, the ambiguity of root cause labeling, and the fragility of reproducibility over time, which, in turn, affect external validity, construct validity, and corpus stability. Although each limitation was mitigated to some extent, they still constrain the scope of the findings and motivate future work on broader corpus construction, inter-rater validation of labels, and more resilient reproduction pipelines.

Selection bias, sample size, and external validity. Because the corpus is sourced from OSS-Fuzz disclosures, it is biased toward memory-safety vulnerabilities in C/C++ programs and toward components amenable to fuzzing, such as libraries. Automated crash reproduction is only feasible for a subset of cases, so the final corpus contains 30 reproduced samples, drawn from 69 reference-matched cases among 1,520 OSV entries with product and CWE overlap in NVD. This selection bias and limited sample size constrain external validity, as the corpus does not reflect the broader distribution of vulnerability classes, languages, or project types. The reproduced set nevertheless preserves the bug-cause distribution from Chapter 2, with *Missing* cases at roughly 67% and *Incorrect* at 33%, partially mitigating concerns about bug composition. The results should therefore be interpreted as characterizing what can be reproduced from publicly available artifacts, rather than as evidence of the full CVE population.

Labeling ambiguity and construct validity. Root cause labeling begins with identifying the bug in the patch and assigning cause and operation labels, but a fix often performs multiple actions, and selecting the operation that most directly explains the vulnerability is a non-trivial interpretive judgment. What further complicates this labeling is the taxonomic scaffolding: operation labels are drawn from BF, which does not yet cover all operations encountered in practice, and when one

falls outside its scope, CWE is used by mapping the closest weakness class to the corresponding BF operation. CWE itself, however, does not consistently distinguish cause from consequence, so this alignment can be interpretive in some cases. To reduce the risk of inaccurate or inconsistent labels, ambiguous cases were resolved through multiple rounds of AI-assisted analysis, each case was documented with explicit notes on the rationale for the root cause, and consistency checks were applied across cases of the same kind. Even so, alternative interpretations remain possible, and the main threat is the absence of an inter-rater agreement check on the labels.

Infrastructure fragility and corpus stability. Reproduction depends on external artifacts, including OSS-Fuzz infrastructure, repositories, build dependencies, toolchains, and vulnerability reports. The benchmark mitigates this fragility by containerizing the software and execution artifacts needed to reproduce each case with Docker, but rebuilding those containers in the future still depends on artifacts that can change, disappear, or become inconsistent over time. This fragility constrains the stability of the reconstructed corpus and therefore the external validity of the findings derived from it. The results should therefore be interpreted as depending on the state of the OSS-Fuzz ecosystem at the time of reproduction.

3.6.4 Future Work

This chapter suggests several directions for future work. Extending the reproduction pipeline to projects in other programming languages would reduce selection bias and improve the generalizability. Formalizing the AI-assisted root-cause labeling into an automated classification and validation workflow would improve scalability and strengthen label consistency. Enhancing OSV–CVE linkage and improving root-cause localization would further reduce manual effort in corpus construction and analysis. Together, these directions would broaden the scope of the benchmark, improve the reliability of its labels, and make the pipeline more scalable.

Extending vulnerability coverage. One promising direction is to extend the pipeline to additional projects, programming languages, and vulnerability classes by building on OSS-Fuzz's support for multi-language project integration. In practice, this would require creating Docker-based build environments and sanitizer configurations to integrate suitable projects with fuzz targets, and then adapting the current workflow to run fuzzing against vulnerable revisions. Such an extension would reduce selection bias in the current corpus, broaden coverage across languages and project types, and improve generalizability beyond the current OSS-Fuzz slice.

AI-assisted labeling and validation. A natural extension is to formalize and automate the AI-assisted analysis used during root-cause labeling, so that both the labeling and its validation can be performed with AI. On the labeling side, large language models could be prompted to assign cause and operation labels to the patch, suggest the closest CWE when no direct match exists in the current BF, and identify the main operation in patches with multiple changes. The manually

labeled cases from Chapter 2 and the benchmark assembled here provide well-defined input–output pairs that could serve as few-shot context or fine-tuning data for this classification task. On the validation side, the lack of inter-rater agreement is the main threat to construct validity; AI systems could address this by acting as independent raters, producing a second set of labels against which agreement can be measured. AI-assisted labeling would reduce manual effort and improve scalability, while AI-assisted validation would provide an objective consistency check.

Automation for linkage and localization. One promising direction is to improve automation in corpus construction and root-cause localization. OSV–CVE linkage currently depends on reference overlap, which is only available for a small fraction of entries; incorporating affected product versions as a fallback matching criterion could increase corpus coverage. In parallel, fault localization with guided fuzzing could reduce the manual effort required to identify the root cause by generating test cases from the crashing input that leads to the location where the program deviates from the proper state. Together, these extensions would make the pipeline more scalable, improve corpus coverage, and reduce the manual effort required to localize the root cause of vulnerabilities.

3.7 Summary

Objective. Vulnerability benchmarks typically represent how each instance manifests, for example as a crash, a sanitizer report, or an error location. Because the underlying bug is rarely recorded explicitly, a benchmark built solely on manifestations does not enable evaluations to distinguish the bug from its observable consequences. This chapter constructs a benchmark in which each instance preserves both the bug and the error as distinct elements of the vulnerability, following the bug $\rightarrow$ error $\rightarrow$ failure representation introduced in Chapter 2. Concretely, the chapter addresses *RQ2: which vulnerability attributes must a benchmark preserve for reliable evaluation?*

Method. A pipeline was developed to transform OSS-Fuzz disclosures into executable, validated benchmark instances by sourcing qualified OSV–CVE pairs, reconstructing historical build contexts, reproducing crashes from fuzzing test cases, and localizing both the bug and error locations from crash reports and fixing patches. Reproduced cases are validated by checking that crash reports match OSS-Fuzz disclosures and that localized bugs are consistent with fixing patches, with manual confirmation. Each instance records the error operation and its mapped CWE at the crash location, and the root cause label and its mapped CWE at the bug location, following the schema introduced in Chapter 2.

Findings. Reproduction is feasible but bounded: of 69 linked CVEs, 40 build successfully and 30 reproduce with crash-matching reports, with the remaining cases failing due to practical engineering constraints rather than any observable pattern in vulnerability type. In the reproduced core, bug and error locations frequently diverge: 13 of 30 cases cross function boundaries with call-graph distances up to nine edges, and same-function cases show line-level separation up to

155 lines. This separation shows that bugs and errors are positionally distinct, a structure that the assessment complements: error operations concentrate in two memory usage types and a small set of CWEs, while bug operations span eight distinct types across a disjoint set. This assessment shows that bugs and errors are semantically distinct even when they coincide, a relationship that cross-referencing further articulates: bug operations vary with separation — same-function cases cluster on *Missing* DATA CHECK, while outside-same-function cases shift toward *Incorrect* DATA TYPE and MEMORY — while error operations remain concentrated across all separation categories. This asymmetry shows that manifestation alone does not provide sufficient information to distinguish one vulnerability from another, and that a benchmark must record both locations and characterize the bug in terms of cause and operation, thereby answering RQ2.

Implications. Because bug and error locations do not always coincide, empirical evaluations must distinguish whether a tool matches the bug location or the error location and report which ground truth is used when measuring tools, since evaluations limited to error locations may overstate detection and assessment capabilities. Because bug operations span diverse types while error operations concentrate on memory usage, benchmarks and tools must support both bug and error localization and classify the operations that characterize each stage of the chain, with bug localization remaining the least addressed of the two. Because reproduction depends on project availability and root-cause labels are derived from patches, any benchmark assembled this way reflects what maintainers provide and sustain, and should be treated as a snapshot rather than a stable ground truth.

Chapter 4

Evaluating ML-Based SAST

4.1 Introduction

This chapter motivates the need to evaluate SAST tools at the root cause level and outlines how the chapter addresses it. Section 4.1.1 frames the practical gap: SAST tools over-report and are often assessed with oracles that describe the manifestation side of vulnerabilities rather than their underlying bug [73, 43, 186, 140, 30, 258, 94]. Section 4.1.2 summarizes the evaluation and triage design, combining the benchmark from Chapter 3 with a reasoning layer and ML-based triage over SAST warnings. Section 4.1.3 states the central research question on ML-based triage for actionable warnings and presents the chapter's contributions in the context of prior work on SAST evaluation and warning classification [97, 228, 243, 113, 247, 71, 72, 89, 117].

4.1.1 Motivation and gap

SAST tools can scan large codebases and flag suspicious code patterns, but their outputs are often difficult to use in practice. Warnings are frequently noisy, imprecisely localized, and underspecified about what the code is doing. A warning is actionable only when it helps determine that the flagged code is worth fixing. In practice, low precision forces developers to spend time investigating false positives, delays remediation, and reduces confidence in tool outputs, while missed bugs limit the effectiveness of SAST tools. Improving actionability, therefore, requires both understanding what tools actually detect and reliably identifying which warnings are worth fixing.

A first gap is that evaluations of SAST tools often measure detection on vulnerability manifestations rather than on the bug itself, and they do not always evaluate whether matched warnings are semantically correct. Vulnerabilities propagate through a chain from bug to error to failure, yet many studies define their primary oracle at the level of the error or use coarse granularities, such as files or functions, that blur the distinction between root cause and manifestation. Most studies also assess matched warnings using reported CWE labels grouped into coarse abstraction categories, which remain tied to the manifestation side of the vulnerability. As a result, current evaluation practices do not clearly show whether tools detect the bug or only its observable manifestations, nor whether matched warnings actually characterize what is wrong at those locations.

A second gap is that recent actionable warning identification methods address limitations such as manual feature engineering, limited supervision, and weak generalization, yet Pre-Trained Transformer Models (PTM) and Large Language Models (LLM) based approaches remain underexplored and treated independently. LLM approaches can reason about warnings in code context and synthesize relevant patterns into natural language explanations approximating expert semantic analysis, but remain vulnerable to hallucination, threatening the correctness and reliability of their outputs. PTM based classifiers learn generalizable representations during pre-training and can be fine-tuned for specific downstream tasks with limited labeled data, but their use for actionable warning identification remains limited. Existing work, therefore, points to two complementary capabilities not yet combined: LLMs can articulate the semantic patterns relevant to a warning as structured natural language explanations, while PTMs can be fine-tuned on those explanations to learn a decision function for distinguishing actionable from unactionable warnings.

4.1.2 Evaluation, triage approach, and scope

To address the gap between what evaluations measure and what tools are expected to detect, this chapter compares the detection and assessment of SAST tools under both bug and error oracles, using the benchmark introduced in Chapter 3. This allows two aspects of tool behavior to be measured: (i) exact line detection under each oracle, and (ii) whether the CWE-derived operation in warnings that match benchmark locations conveys the expected operation recorded in the benchmark. The evaluation is performed on four build independent SAST tools selected from the related work: `Cppcheck`, `Flawfinder`, `Semgrep`, and `CodeQL`. Selection is constrained by practical requirements: analyzers must run without build reproduction, support the vulnerability classes represented in the benchmark through their available rule sets, and emit SARIF natively. All tools are executed under controlled configurations in containerized environments with fixed parameters, so that differences in results reflect analyzer behavior rather than environmental variation.

To address warning actionability, a reasoning layer reinterprets each warning in terms of code behavior by generating explanations that serve as input to a lightweight text classifier, whose outputs are used to filter warnings by actionability. The explanations are generated with a commercial LLM from the code context at the warning location and the corresponding tool message. These explanations identify the relevant operation and qualify it as proper, missing, or incorrect. The classifier is based on a PTM fine tuned on a corpus built from CVE fix commits. For each retained case, diff analysis of the patch identifies a canonical bug location, and the same SAST tools are run on both the vulnerable and fixed revisions. Warnings are then labeled by location overlap alone: warnings that overlap the bug location in the vulnerable revision are labeled as correct, while warnings that overlap the same location in the fixed revision are labeled as incorrect. Both training and evaluation operate on warnings for which the reasoning layer produces explanations, and evaluation is further restricted to files containing at least one ground truth match.

The scope is therefore constrained by the benchmark, the selected tools, and the training corpus: reproducible fuzzing derived C/C++ memory safety vulnerabilities, CVEs with manifestation related to the CWE categories represented in the benchmark, and four lightweight analyzers whose

rule support is sufficient for this setting. As a result, the conclusions primarily reflect this setting rather than the full range of vulnerability classes, programming languages, or deployment contexts.

4.1.3 Research question and contributions

The preceding subsections identify a persistent gap between how SAST tools are evaluated and how their warnings are used in practice. If detection is to reflect bug localization rather than incidental proximity to the error, and if triage is to retain warnings aligned with the bug rather than merely reduce volume, then ML-based triage must be examined under an evaluation that distinguishes the bug from the resulting error. This motivates the chapter's research question: **RQ3. To what extent can ML-based triage identify actionable SAST warnings?**

> In practice, warnings may align with the error rather than the bug, and their reported operations are often too coarse to confirm what is wrong at the flagged location. Triage based on richer descriptions of warnings could reduce false positives while preserving warnings that match the bug, making high-volume static analysis more practical.

Building on the vulnerability representation introduced in Chapter 2 and the benchmark constructed in Chapter 3, this chapter first evaluates what SAST tools detect under the two reference locations, and then evaluates whether ML-based triage improves the actionability of those outputs. The following contributions address these two stages:

- **Evaluation of SAST tool detection at bug and error locations** to determine whether tools tend to detect the root cause or its manifestation. Recall is higher at error locations (up to 13.8%) than at bug locations (up to 7.1%), with precision below 1% across all tools.
- **Evaluation of SAST tool assessment at operation level** to determine whether location-matching warnings correspond to bug or error operations. Location-matching warnings do not correspond to the recorded operations, which makes them predominantly incidental.
- **A reasoning layer for SAST tools** that enriches SARIF reports with compact natural language explanations generated by a commercial LLM, which identify and qualify the operation and operand in the code at each warning location, to support warning actionability.
- **ML-based triage over explanations of warnings** produced by the reasoning layer, used to train a lightweight classifier to assess warning actionability. At bug locations, triage preserves all true positives while removing 40 false positives (from 110 to 70), while at error locations, it removes 415 false positives (from 873 to 458) while losing 4 true positives.

These findings are bound by the size of the benchmark and the selected tool set, rendering the reported metrics indicative rather than statistically robust. Extending the evaluation to a broader tool set and benchmark is the most direct path toward more robust conclusions. Nonetheless, the results show that reporting detection separately at bug and error locations is necessary to avoid

overstating tool capabilities, and that natural-language explanations of warnings constitute a viable representation for learning a decision function for actionability.

4.1.4 Publications context and roadmap

This chapter builds on several prior publications that informed its methodology and framing without directly reusing artifacts or empirical results. Two papers on Transformer-based vulnerability classification in Java establish the feasibility of pretrained encoders for vulnerability related discrimination tasks and motivate the chapter's use of a lightweight Transformer classifier for actionable warning identification [147, 148]. The follow-up study further shows that generalization is strongly shaped by dataset quality and construction choices, including realistic samples, label balance, normalization, and root cause localization, which influenced this chapter's emphasis on careful dataset curation and supervision at the bug location [148].

A complementary line of work, TENET, provides the labeling strategy used in this chapter by combining diff derived vulnerable and fixed locations with static analysis outputs on both code versions, extending its original vulnerability detection setting to actionable warning discrimination [192]. Finally, experiments evaluating large language models on vulnerability detection and repair in C indicate that LLMs can associate vulnerability classes with code constructs and reason over local context, motivating this chapter's use of compact explanations of warnings as a reasoning layer over heterogeneous SAST warnings and as input to downstream triage [60].

Roadmap. The remainder of this chapter is organized as follows. Section 4.2 introduces background on SAST, ML, and LLM. Section 4.3 surveys related work on empirical SAST evaluation and actionable warning identification, motivating the design choices adopted here. Section 4.4 details the methodology, including the evaluation design, training data construction, reasoning layer, and triage model. Section 4.5 presents detection, assessment, and triage results on the benchmark. Section 4.6 interprets these findings, discusses implications and limitations, and outlines future directions, and Section 4.7 concludes with a concise summary of the chapter's contributions.

4.2 Background

This section outlines the technical foundations underlying the chapter's evaluation and triage framework. Section 4.2.1 introduces definitions of static analysis, followed by Section 4.2.2, which focuses on static analysis techniques specialized for vulnerability detection and their operationalization in SAST tools. Section 4.2.3 then presents machine-learning approaches in static analysis, covering both vulnerability detection and actionable-warning identification. Finally, Section 4.2.4 introduces pretrained Transformer architectures and large language models, which form the basis for modern transfer-learning and generative reasoning approaches in code analysis.

4.2.1 Static Analysis: Definitions and Foundations

Static analysis is a seminal program analysis approach that has evolved from a basic compiler optimization technique into a central component of software correctness and verification. Its primary goal is to efficiently compute reliable approximations of a program's behavior without executing it. F. Nielson *et al.* [174] provide an overview of several static, compile-time approaches that form the foundation of modern analysis techniques, including:

- *Data Flow Analysis*: a family of techniques that interpret a program as a graph and derive equations or constraints describing the flow of data along possible execution paths;
- *Constraint-Based Analysis*: an approach that formulates properties of interest as constraints and uses formal methods to determine whether those constraints are satisfied by the program;
- *Abstract Interpretation*: a framework for computing sound approximations of program behavior by interpreting programs over abstract domains that approximate sets of concrete values or states.

Subsequent surveys, such as D'Silva *et al.* [44], extend this landscape to encompass formal verification–oriented analyses, including model checking and variants of abstract static analysis. These techniques build symbolic models of program behavior and explore program states against formal specifications. While powerful, such approaches tend to be computationally intensive and rely on complete program models or build environments. Complementary classifications, such as those by Gosain and Sharma [83], also include lightweight techniques such as *syntactic pattern matching*, which detects potentially dangerous constructs through token or AST pattern queries, alongside more heavyweight techniques such as *symbolic execution* and *theorem proving*. Together, these taxonomies illustrate the breadth of static analysis methodologies, spanning from shallow syntactic heuristics to deep semantic verification.

Static analysis encompasses a broad methodological space, ranging from graph-based data flow analyses to constraint-based methods and abstract interpretation. The overview provided in this subsection outlines the foundational principles that underlie these approaches. For this work, the most relevant dimensions are those that directly shape how tools surface security-relevant defects in real-world workflows.

4.2.2 Static Analysis for Vulnerability Detection

Techniques used for vulnerability detection largely derive from and specialize in the foundational static analysis principles introduced earlier. Reflecting this lineage, H. Shahriar and M. Zulkernine [203] classify vulnerability-oriented static analyses into four inference types: *Tainted Data-Flow-Based*, *Constraint-Based*, *Annotation-Based*, and *String-Pattern-Matching-Based*. *Tainted Data-Flow-Based* techniques extend classical data-flow analysis to track untrusted or malicious inputs

through program variables and detect unsafe propagations. *Constraint-Based* techniques, including symbolic execution, integer range analysis, and demand-driven constraint solving, identify program states that violate safety properties by solving path or value constraints. *Annotation-based* techniques leverage source-level specifications to verify program behavior and share conceptual ties with model checking, especially in their use of specified properties as verification targets. *String-Pattern-Matching-Based* techniques operate over tokenized code or abstract syntax trees to search for syntactic signatures of dangerous constructs, such as unsafe memory operations, unchecked inputs, or missing error handling. Through these adaptations, vulnerability detection applies the broader spectrum of static analysis techniques to the specific task of identifying security-relevant program behaviors.

Table 4.1: Static Analyzers used in Empirical Studies [8, 50, 140]

Name	**Year**	**Technique(s)**	**Programming Language(s)**
PCLint [215]	1985	flow-of-control analysis, inter-statement value tracking	C/C++
Klocwork Insight [67]	2001	inter-procedural data flow analysis, symbolic logic	C/C++, C#, and Java
UNO [100]	2001	path-sensitive inter-procedural data-flow analysis	C
Flawfinder [235]	2001	context-sensitive analysis, string pattern matching	C/C++
Coverity Prevent [197]	2002	inter-procedural data flow analysis, statistical inference	C/C++, Java, and C#
Splint [56]	2002	annotations, flow-sensitive and data flow analysis	C/C++
CodeSonar [87] GrammaTech	2005	inter-procedural path-sensitive flow of control analysis	C/C++, C#, Java, and x86
Fortify SCA [33]	2006	inter-procedural taint analysis, control flow sensitive	18 (C/C++, C#, Java, PHP, etc.)
CxSAST [104] Checkmarx	2006	query-based and tainted data flow anlysis	9 (C/C++, C# Java, PHP, etc.)
Goanna [63]	2006	tree-pattern matching, data flow analysis, model checking	C/C++
CodeQL [164]	2006	query-based and tainted data flow analysis	C/C++, C#, Java, JS,
Frama-C [121]	2007	abstract interpretation, deductive verification, and concolic testing	C
Clang SA [220]	2007	annotations, symbolic execution, taint inter-procedural analysis	C, C++, and Objective-C
Cppcheck [151]	2009	context-sensitive control/value flow analysis	C/C++
Infer [26]	2011	shape analysis, Hoare logic, inter-procedural analysis	Java, C/C++, Objective-C

Static analysis in practice: To detect security-relevant weaknesses and address the increasing prevalence of software vulnerabilities, static analysis techniques such as syntactic pattern matching, data-flow reasoning, and constraint-based analyses have been operationalized by industry vendors, open-source communities, and academic researchers into tools known as SAST [83, 203, 176, 179]. Such tools are intended for integration into development workflows and are commonly applied during coding and early testing stages [43]. In practice, SAST tools analyze source code (or intermediate representations) without execution and emit *warnings* when predefined rules or inferred safety properties are violated. A warning typically includes a source location (e.g., file and line), a rule identifier, a textual diagnostic message, a severity level, and often a mapped weakness category such as a CWE. Reports are commonly exported in standardized formats (e.g., SARIF) to facilitate integration with continuous integration pipelines and issue trackers. Depending on their underlying technique, tools may detect syntactic patterns, unsafe data flows, constraint violations, or infeasible path conditions, but in all cases the output is a set of static findings that require human or automated triage.

First-generation analyzers. Table 4.1 summarizes representative analyzers and the techniques they implement, illustrating how successive generations of tools have incorporated increasingly sophisticated methods. Early shallow analyzers, such as PC-Lint [215], relied primarily on syntactic information and simple intra-procedural checks. These first-generation tools were effective at detecting local syntactic and semantic defects—unused variables, unreachable code, inconsistent declarations—but had limited ability to reason about runtime behavior or inter-procedural security-relevant conditions.

Second-generation analyzers. During the early 2000s, static analysis tools evolved beyond simple syntactic checks by incorporating inter-procedural control-flow analysis and richer data-flow reasoning to approximate possible runtime values and exclude infeasible execution paths. These advances enabled detection of inter-modular faults with security implications that earlier tools could not capture. Representative tools reflecting these developments include Klocwork Insight [67], Flawfinder [235], Coverity Prevent [197], Splint [56], CodeSonar [87], Fortify SCA [33], and CxSAST [104].

Third-generation analyzers. Beyond the mid-2000s, subsequent generations integrated automated formal verification techniques to strengthen detection guarantees. These approaches incorporate symbolic reasoning, abstract interpretation, and model-checking–based methods that over-approximate program behaviors and aim for soundness with respect to specific vulnerability classes. While capable of providing stronger assurances, such analyses typically require more computational resources, formal specifications, and complex build configurations. Representative tools in this category include Goanna [62], Frama-C [121], Clang Static Analyzer [220], and Infer [26].

4.2.3 Machine-Learning in Static Analysis and Vulnerability Detection

Recent advances in machine learning have introduced data-driven approaches that operate over static representations of source code and infer vulnerability-related properties without program execution, positioning ML as a next-generation form of static analysis. Two closely related paradigms have emerged: (i) *vulnerability detection*, which classifies code fragments, functions, or files as vulnerable or non-vulnerable, and (ii) *actionable-warning identification*, which classifies or ranks static-analysis warnings according to whether they correspond to actionable defects. Systematic studies document the rapid growth of ML in security [110], and existing taxonomies consistently group approaches into *traditional machine learning* and *deep learning* categories [92, 217, 89], distinguishing between models built on handcrafted features and those that learn representations directly from token sequences, syntax trees, or program graphs [74]. This dichotomy applies to both paradigms, which follow a common supervised pipeline of data collection, preprocessing and encoding, model training, and evaluation [217, 72, 92]. While vulnerability detection predicts properties of code artifacts, actionable-warning identification predicts properties of analyzer outputs—an important distinction given that SAST tools generate large volumes of warnings, many unactionable [96, 94], and that default severities often misalign with developer priorities [120]. Despite differing targets, both settings rely on static program representations and supervised learning over labeled artifacts, constituting overlapping data-driven extensions of traditional static analysis.

Traditional machine-learning methods: In both *vulnerability detection* and *actionable-warning identification*, early ML approaches predominantly relied on supervised learning with manually engineered feature vectors. For *vulnerability detection*, these methods typically model binary prediction tasks using software metrics and project metadata as inputs, including complexity measures, code churn, developer activity, execution metrics, or vulnerability history [255, 208, 209]. Common learners include decision trees, support vector machines, regression models, Bayesian classifiers, and ensemble techniques [208, 166, 200, 165, 205, 181, 187]. In *actionable-warning identification*, a similar paradigm emerged in which warnings are represented through handcrafted attributes derived from warning metadata (e.g., type, severity, location) and surrounding code characteristics (e.g., file size, path conditions, call counts) [97, 249, 91, 228]. Although these classifiers demonstrate that statistical discrimination of vulnerable artifacts or actionable warnings is feasible, their effectiveness depends heavily on domain-specific feature design, and empirical reassessments show that performance can degrade under cross-project settings [243, 113, 71, 247]. In both paradigms, program behavior is reduced to fixed, human-designed feature spaces, limiting semantic expressiveness and robustness under distributional shift.

Deep-learning methods: Deep-learning approaches replace manual feature engineering with neural architectures that learn representations directly from raw program artifacts. In vulnerability detection, early work from around 2017 onward adopted recurrent, feed-forward, and graph-based

neural networks to model token sequences, instruction traces, control-flow graphs, and other structural encodings of code [217, 185, 134, 182, 188, 260]. These models automatically induce vector embeddings that capture syntactic and structural regularities associated with vulnerable behavior. A parallel shift occurred in actionable-warning identification, where neural classifiers replaced handcrafted warning features with learned embeddings derived from source code, abstract syntax trees, or control/data-flow graphs [123, 129, 247, 244]. Across both paradigms, representation learning centers on transforming code or warning contexts into dense vector spaces optimized jointly with the classification objective. Although these models eliminate manual feature design and often improve in-distribution accuracy, empirical evidence indicates that their learned representations remain strongly tied to the training data and task formulation, raising concerns about robustness and generalization under domain shift [247, 244, 72].

4.2.4 Pretrained Transformers and Large Language Models

The progression from task-specific deep neural networks to pretrained Transformer architectures marks a structural shift in data-driven static analysis. Introduced in 2017 [223], the Transformer replaces recurrent computation with self-attention mechanisms that allow each token to weigh the relevance of other tokens in the sequence and to still model long-range dependencies. Its encoder–decoder architecture can be configured in three modes: encoder-only models for representation learning and classification, decoder-only models for autoregressive generation, and full encoder–decoder models for sequence-to-sequence tasks [139]. By combining multi-head self-attention with positional encoding and feed-forward layers, Transformers learn contextualized token representations that capture lexical, syntactic, and structural relationships in sequences. This architectural flexibility enabled large-scale pretraining on massive corpora, decoupling representation learning from downstream supervision and establishing transfer learning as a central paradigm in natural language processing and, increasingly, in code analysis.

Pretrained models: PTM, such as BERT [49], are trained on large unlabeled corpora using self-supervised objectives (e.g., masked language modeling) to learn general-purpose contextual embeddings, which can then be adapted to downstream tasks via fine-tuning [180]. Variants such as RoBERTa [143], DistilBERT [199], and code-oriented models including CodeBERT [64] extend this paradigm to domain-specific settings. In vulnerability detection, Ziems and Wu [262] demonstrated that fine-tuning BERT on C/C++ code achieves substantially higher accuracy than LSTM-based baselines for file-level vulnerability classification. Similarly, in the identification of actionable warnings, Ge et al. [72] evaluated multiple pretrained architectures (encoder-only, decoder-only, and encoder–decoder) and showed consistent improvements over traditional deep learning models when fine-tuned on warning datasets. Across both paradigms, pretrained models shift the learning burden from task-specific representation induction to adaptation of broadly learned semantic embeddings, improving performance particularly when labeled data is limited.

Large language models: Scaling Transformer architectures in model size and training data led to the emergence of LLM, which exhibit zero-shot (performing tasks without task-specific training) and few-shot (conditioning on a few examples in the prompt) generalization capabilities [23, 256]. Unlike smaller pretrained models that require explicit fine-tuning, LLMs can perform downstream tasks through prompt-based instruction, effectively treating vulnerability detection or warning triage as conditional text-generation problems. Empirical studies show that models such as GPT-3.5 and GPT-4 achieve competitive or superior performance relative to code-specific pretrained models in vulnerability detection under well-designed prompting strategies [259, 29, 240]. In the context of static-analysis triage, LLMs have been integrated with SAST outputs to reason about warnings, extract relevant code context, and generate structured decisions or explanations [116, 233, 105, 130]. While LLMs expand static analysis from classification toward generative reasoning, their effectiveness depends on prompt design, contextual scope, and the reliability of generated outputs, distinguishing them from fine-tuned encoder-based models.

4.3 Related Work

This section reviews prior research along four dimensions relevant to this chapter. Section 4.3.1 surveys empirical studies that evaluate SAST tools for C/C++ vulnerability detection, highlighting differences in benchmark construction, oracle definition, and matching granularity. Section 4.3.2 examines approaches for identifying actionable warnings, spanning feature-engineered, representation-learning, and transfer-learning classifiers. Section 4.3.3 discusses generative reasoning–based warning triage, where large language models are used to produce explanations, infer specifications, or directly assess warning validity. Finally, Section 4.3.4 synthesizes these threads and articulates the remaining research gap that motivates the chapter's causality-aware evaluation and ML-based triage perspective.

4.3.1 Empirical Studies of SAST Tools for C/C++ Vulnerability Detection

Static analysis tools for C/C++ have been studied extensively, but reported effectiveness varies widely across the literature—often because studies operationalize "vulnerability detection" in fundamentally different ways. Prior evaluations differ in the benchmarks they use (synthetic test suites versus real CVEs), in what they treat as ground truth (injected weakness locations, patch-induced deltas, or curated fault metadata), and in how they match tool warnings to that oracle (line-, function-, or file-level proximity, sometimes with coarse CWE grouping). These choices matter: they can make tools appear accurate when they merely warn near a crash site, or, conversely, make tools appear to "miss" vulnerabilities when they report manifestations rather than root causes. To clarify how these methodological decisions shape observed results, we organize empirical studies of SAST tools for C/C++ into three categories: evaluations on synthetic benchmarks. These hybrid designs mix synthetic suites with small real-world case studies, and CVE-centric benchmarks based entirely on production vulnerabilities.

Synthetic benchmarks: Evaluations based on artificial code are comparatively straightforward, as synthetic benchmarks are constructed a priori around known weakness patterns, provide explicit ground-truth annotations, and involve programs that are substantially simpler than production codebases; however, their controlled nature and non-representative vulnerability distributions limit their ability to reflect real-world SAST performance [73, 186]. The Juliet Test Suite (JTS) is a prominent example, offering paired "good" and "bad" variants with metadata identifying injected weaknesses and source locations. Despite these favorable conditions, multiple studies report limited detection capability: Díaz and Bermejo [50] found that none of nine SAST tools detected all 78 C/C++ test cases, Wagner and Sametinger [225] observed similarly low detection rates on two open-source SAST tools, Goseva-Popstojanova and Perhinschi [84] reported no statistically significant differences among three commercial tools, and Kaur and Nayyar [114] identified systematic blind spots—such as null pointer dereferences—when comparing Flawfinder [235], RATS, and Cppcheck [151]. Beyond JTS, Arusoaie et al. [8] evaluated ten open-source C/C++ analyzers on 639 synthetic defects from the Toyota ITC suite, where detection is defined as any warning on a pre-annotated line, regardless of semantic alignment with the defect type; while this enables scalable comparison, it conflates manifestation sites with root causes and allows noisy tools to score chance hits, particularly for complex defect classes. Dubniczky et al. [54] assessed thirteen tools on a hand-crafted dataset of 250 compact C programs covering 25 CWEs, reporting high false-positive rates and substantial manual validation effort. Taken together, these studies demonstrate that synthetic benchmarks are effective for assessing whether tools can flag known weakness manifestations in controlled settings, but their ground truth rarely captures root-cause bugs in the stronger sense of the earliest missing or incorrect code that initiates a vulnerability's propagation chain.

Hybrid evaluations: Several studies attempt to mitigate the limitations of purely synthetic benchmarks by complementing them with small-scale evaluations on real-world CVEs, while still relying primarily on artificial test suites. Gentsch [73] evaluates eleven open-source SAST tools mainly on the Juliet Test Suite 1.3 for C/C++, counting a warning as a true positive if it intersects the annotated fault location without requiring agreement with the manifest's CWE label, and then conducts a secondary experiment on a CVE-bearing version of Wireshark (SARD-94). While tools perform noticeably better on Juliet, only 17 of 83 CVEs are detected in the production code, and manual inspection is required to distinguish true vulnerability hits from warnings triggered on surrounding crash-context locations. Croft et al. [43] similarly combine Juliet with pre-patch file versions associated with CVE-fixing commits in large OSS projects, but evaluate detection primarily at file granularity, treating any warning in a vulnerable file as a hit. Their manual analysis shows that most warnings are incidental and rarely localize the true vulnerable statements, which helps explain both the high false-positive burden and the limited diagnostic value of such evaluations. Li et al. [136] extend this hybrid paradigm by comparing conventional and query-based SAST tools on Juliet and a curated benchmark of 136 CVEs, measuring instance-level detectability rather than source-level localization. Although they report superior

performance of query-based tools and demonstrate that targeted query refinements can substantially improve results, their findings also reveal that measured detectability is highly sensitive to benchmark construction and query quality, rather than to a tool's ability to precisely identify the locations of real vulnerabilities. Taken together, these hybrid evaluations consistently show that performance observed on synthetic benchmarks does not reliably transfer to production code and that coarse-grained detection metrics obscure what SAST tools actually identify.

CVE-centric benchmarks: A growing body of empirical work focuses exclusively on real vulnerabilities in natural C/C++ code, aiming to assess whether SAST tools detect CVEs in settings closer to production practice. Pereira and Vieira [186] analyze Mozilla vulnerabilities by running Cppcheck and Flawfinder on pairs of pre-fix and post-fix snapshots, operationalizing detection via a file-level alert-delta heuristic: a vulnerable file is considered detected if the number of warnings changes after the fix. While scalable, this approach measures warning churn rather than whether a tool identified the correct vulnerability location and type, inflating reported detection rates and entangling results with unrelated alert noise. Lipp et al. [140] advance this line of work with a large CVE-grounded benchmark spanning 27 OSS projects and 192 vulnerabilities, aggregating CVE reports, patches, and—in part—the Magma dataset's explicit fault/manifestation metadata. To accommodate the fact that analyzers often flag manifestation sites rather than root causes, they adopt function-level evaluation and map tool warnings to grouped CWE classes. Their results show that individual tools miss a substantial fraction of CVEs, while tool combinations improve detection at the cost of increased review burden, underscoring a precision–coverage trade-off. Charoenwet et al. [30] study SAST tools in the context of secure code review, using vulnerability-contributing commits (VCCs) linked to CVEs, evaluating detection at the file and function granularities, and measuring practical review benefits under constrained budgets. Although SAST alerts can improve prioritization, most warnings in vulnerable functions are not directly related to the vulnerability change, and many VCCs receive no relevant warnings at all. Zhou et al. [258] similarly evaluate function-level detection on a smaller CVE benchmark and report low detection ratios for individual tools, with modest gains from tool combinations accompanied by increased code flagged. Hassler et al. [94] further refine evaluation by requiring CWE-consistent warnings match vulnerable functions, showing that semantic analyzers outperform syntactic ones in type-aligned detection; however, their qualitative analysis reveals that even CWE-matched reports frequently fail to pinpoint the true root-cause lines.

Table 4.2 consolidates the methodological design choices of empirical studies that evaluate SAST tools on real C/C++ CVEs, revealing substantial variation—and recurring ambiguity—across benchmarks, evaluation targets, and matching strategies. Across studies, a common pattern emerges in which two distinct evaluation objectives are implicitly interleaved. In detection-oriented scenarios, a vulnerability is considered detected if a tool emits any warning at or near a vulnerable artifact (e.g., file or function), regardless of whether the warning correctly characterizes the vulnerability. In assessment-oriented scenarios, stricter matching is imposed—typically requiring alignment in location and vulnerability type—which often results in markedly lower

Table 4.2: Summary of Empirical Studies that Evaluate SAST Tools on Real C/C++ CVEs

Study	Benchmark		Evaluation			
	Source	#CVE	Target	Granularity	Matching	Metrics
Pereira & 2020 [186]	Mozilla [4]	2441	Error	File	Location + CWE Group	Prec., Rec., F1-Score
Gentsch 2020 [73]	SARD [219]	85	Error	Line	Location	Prec., Rec., MCC
Croft et al. 2021 [43]	3 Projects from [111]	1593	Error	File/Line	Location + CWE Group	Prec., Rec., MCC
Lipp et al. 2022 [140]	Magma [95] and [77, 66]	192	Mixed	Function	Location + CWE Pillar	Custom ratios
Zhou et al. 2024 [258]	6 Projects from [140]	85	Error	Function	Location + CWE Type	Ratios from [140]
Charoenwet et al. 2024 [30]	92 Projects from [102]	319	Error	File / Function	Location + CWE Pillar	Prec., Rec., IFA [101]
Li et al. 2024 [136]	51 Projects from [168]	136	Error	Instance	Location + CWE Type	Accuracy
Hassler et al. 2025 [94]	Magma [95]	112	Mixed	Function	Location + CWE Group	TPs, FPs, FNs

Mixed - intermingles fault-like and error-like locations across defect types

performance. This divergence reflects the fact that tools may successfully flag suspicious code regions while failing to accurately explain why they are vulnerable. Furthermore, most studies define their primary oracles at the level of error manifestations rather than root-cause faults, frequently adopting coarse-grained granularities (files or functions) and location-based matching, sometimes augmented with CWE grouping. While such abstractions make large-scale evaluation tractable, they blur the distinction between causal detection and incidental warning proximity. Moreover, assessment metrics struggle to capture false positives: evaluating whether a warning is "incorrect" requires confidence that the surrounding code is otherwise vulnerability-free—an assumption that rarely holds for complex, evolving codebases. As a result, some studies measure both detection and assessment under multiple scenarios, but the reported metrics conflate tool limitations with oracle imprecision and semantic mismatch. More generally, empirical SAST evaluations must state explicitly whether their ground truth represents fault sites, error states, or failure symptoms, and whether this definition aligns with what static analyzers are designed to report; otherwise, apparent performance deficits may reflect evaluation artifacts rather than genuine analytical shortcomings.

4.3.2 Identification of Actionable Warnings through Data-Oriented Classification

As seen in the previous subsections, empirical studies evaluate SAST tools as automated scanners that surface potential bugs and errors as warnings, but each warning still demands developer judgment to decide whether it indicates an important anomaly worth fixing—an *actionable* warning—or whether it is unimportant/incorrect and thus *unactionable* [96]. In practice, this

creates heavy triage pressure—sometimes on the order of thousands of warnings per vulnerability [94]—while only a small portion of warnings are actionable and the remainder are unactionable false positives [96]. Although tools attach severity/priority levels, early evidence shows that default, category-level priorities are often poorly calibrated to what developers actually fix [120], motivating actionable-warning identification methods that combine SAST outputs with other information about the software under analysis via additional software-artifact "characteristics" [96]. These methods are typically framed as either *classification* (label warnings actionable vs. unactionable) or *prioritization* (rank warnings by likelihood of being actionable) and often rely on mathematical/statistical modelling or machine learning [98, 89, 71].

Feature-engineered classifiers: This first generation of techniques [97, 249, 91] relied on manually crafted feature vectors capturing attributes and metrics derived from warning metadata (e.g., type, severity, and location) and surrounding code (e.g., file size, number of path conditions, and function call counts) [228, 246, 247]. Wang et al. [228] conducted a systematic experimental evaluation of most of the features in the literature of warning identification and identified a common set of 23 effective features (referred to as *Golden Features*). Building on this observation, Yang et al. [243] reported that these Golden Features could be approximated by fewer dimensions and argued that, for such intrinsically low-complexity data, a wide range of learners could achieve comparable performance, leading them to conclude that recognizing actionable warnings is relatively easy. However, Kang et al. [113] later replicated the study by Yang et al. [243] and identified subtle methodological issues in the Wang et al. [228] dataset, including duplicate instances and leaked ground-truth labels, and under corrected conditions, the Golden Features were found to perform only marginally better than random predictions. More broadly, although engineered features can capture warnings exhibiting regular surface-level patterns—often by identifying salient identifiers in the source code—they rely heavily on domain expertise and compress program behavior into a discrete, human-designed feature space, which carries limited semantic signal about (un)actionable warnings and consequently limits generalization to new programs [71, 247].

Representation-learning classifiers: Subsequent techniques [123, 124, 129, 247, 142, 244] increasingly moved toward neural network–based classifiers that emphasize automatic learning of features encoding semantic and lexical patterns from historical data, and which generally achieve higher accuracy than feature-engineered classifiers [247, 244, 89]. In practice, representation learning refers to the automatic induction of vectorized code representations—commonly realized as *embeddings*—either as learned input transformations or as latent features optimized jointly with the classification objective [244]. In this context, a key distinction lies in whether the mapping from raw program artifacts to vectors is *learned* or *fixed*. Many prior approaches adopt learned mappings by treating the source code associated with warnings as natural-language–like token sequences and learning embeddings directly from data, an approach widely instantiated through recurrent architectures such as LSTM-based models [72, 123, 124, 247, 142, 244]. In contrast, other approaches begin from fixed numerical representations derived from textual and non-textual

inputs (e.g., abstract syntax trees and control/data-flow graphs), which are encoded through dedicated models [129, 247, 71]. In practice, learned sequential embedding models are usually trained from scratch and produce static representations, assigning each token a single vector regardless of context. Although more expressive, contextualized embeddings have also been explored, Yang et al. [244] found that static embeddings achieved higher F1 scores across six neural architectures, suggesting that models designed under the assumption of relatively stationary embeddings tend to act as pattern detectors over stable dimensions and may wash out finer-grained contextual variation. Overall, while representation-learning-based classifiers eliminate the need for manual feature engineering and can achieve high accuracy on in-distribution data, empirical evidence [247, 244, 72] suggests that their learned representations remain largely task-specific and closely tied to the training distribution.

Transfer-learning classifiers: More recent techniques have shifted from task-specific representation learning toward *pretrained models*, in which knowledge acquired from large, task-agnostic corpora is transferred to downstream tasks via controlled specialization, most commonly through *fine-tuning* [117, 72]. This transfer is enabled by Transformer architectures, whose self-attention mechanisms learn contextual representations that capture token dependencies and scale effectively when trained under self-supervised objectives on large volumes of unlabeled data [117]. As a result, these models induce a general-purpose representational space that encodes shared semantic and structural abstractions across domains, which can be adapted to specific tasks by updating all or part of the model using limited task-specific labeled data [117, 72]. For classification-oriented tasks, encoder-based Transformer models—designed to produce holistic representations of their inputs—are therefore commonly adopted. A representative example is CodeBERT, which has been pretrained on millions of code–text pairs across several widely used programming languages and whose learned representations have been successfully transferred to actionable warning identification [117, 72, 244, 142]. Despite the strong empirical performance of pretrained Transformer models across many natural language processing and software engineering tasks [246, 72, 117], relatively few studies have investigated their effectiveness for actionable warning identification [89, 71]. Notably, Ge et al. [72] reported comparable performance across within- and cross-project settings, indicating improved generalization and robustness to domain shift relative to previous approaches. This advantage arises because pretrained models internalize programming language syntax, coding idioms, error patterns, and warning semantics before task adaptation, allowing fine-tuning to focus on discriminating actionable from non-actionable warnings. In contrast, representation-learning–based classifiers trained from scratch must learn both representations and task supervision from limited labeled data, making the learning problem substantially harder.

4.3.3 Generative Reasoning–Based Warning Triage

While encoder-based models leverage transfer learning to improve classification through richer representations, a growing body of work [233, 226, 138] instead exploits *decoder-based* and large

language models to *generate explanations and reasoning traces*, using generative outputs—such as code interpretations or rationales—as an explicit signal for warning triage. The key premise is that LLMs can approximate expert review by inspecting static-analysis warnings together with relevant code context and reasoning about whether a reported issue reflects a real defect or an incidental pattern [233]. Existing approaches fall broadly into two groups. The first treats the LLM as an end-to-end analyst that consumes warning metadata and additional context (e.g., code, data flows, patterns) and directly emits a verdict, optionally with a rationale. The second embeds LLM reasoning within a larger pipeline, decomposing triage into subtasks such as specification inference, rule synthesis, or contextual extraction, whose outputs feed downstream analyses. At the same time, these lines of work raise concerns about reliability and explanation quality: hallucinations threaten correctness under complex semantics [241] and plausible-sounding explanations may lack faithfulness to the true causal basis of a decision [145]. Emerging studies therefore emphasize the need for structured, interpretable, and evidence-grounded explanations that are not only linguistically coherent but also operationally useful in developer workflows [145, 146].

LLMs as direct warning analysts: The most straightforward strategy frames actionable-warning identification as a classification task solved directly by an LLM. Munson et al. [169] compare Semgrep with LLM-based assessment on OWASP benchmarks by prompting models with both the SAST warning and the flagged code snippet, observing improved detection at the expense of higher false-positive rates. Keltek et al. [116] construct structured prompts embedding scan findings and target code, augmented through retrieval of vulnerability knowledge (e.g., functionality reports, abstractions, known vulnerabilities); responses include CWE identifiers, rationales, and vulnerable line numbers, with effectiveness closely tied to retrieval relevance. Wen et al. [233] unify warnings from multiple tools, extract calling contexts via dependency analysis, and prompt LLMs to issue a verdict—False Alarm, Real Bug, or Unknown—accompanied by an explanation. Iranmanesh et al. [105] impose stricter guardrails through CWE-specific rubrics, scope constraints, and mandatory JSON schemas, requiring models to return structured decisions and confidence scores. Across these studies, the LLM operates as a reasoning engine that ingests contextualized warnings and directly outputs triage judgments, externalizing expert-like inspection into a single generative step.

LLMs as inference components: To mitigate scalability and reliability challenges, more advanced approaches decompose warning triage into coordinated subtasks in which LLMs serve as inference components within a broader analysis workflow. Li et al. [135] combine static analysis with LLM-inferred taint specifications: candidate APIs are labeled as sources or sinks by an LLM, translated into formal specifications for a taint engine, and resulting alerts are subsequently filtered through LLM-based contextual reasoning. Li et al. [130] address Use-Before-Initialization bugs in the Linux kernel via multi-conversation prompting that extracts initializers and constraints, embeds demonstrative examples, and applies majority voting, illustrating prompt engineering challenges in managing token limits and unreliable responses. Lin [138] employs LLMs to derive

updated source–sink rules and semantic embeddings from rich project context (API documentation, commits, comments), which are then consumed by a downstream classifier for alert ranking. Yang et al. [241] synthesize specialized checkers from historical patches through staged prompting—pattern extraction and checker synthesis—with iterative refinement driven by LLM-based false-positive feedback. In these pipelines, generative reasoning is not the final decision mechanism but a multi-capability component that informs downstream analyses.

4.3.4 Summary and Research Gap

The reviewed literature spans two complementary lines of work. Empirical studies of SAST tools for C/C++ vulnerabilities vary in benchmark design, oracle definition, and matching granularity, leading to heterogeneous conclusions about detection and assessment capability. Synthetic benchmarks offer controlled, line-level annotations but rarely represent realistic bugs, whereas hybrid benchmarks with CVEs introduce realism but rely on coarse oracles that conflate bugs with errors. In parallel, research on actionable-warning identification has progressed from feature-engineered classifiers to representation learning, transfer learning with pretrained encoders, and, more recently, generative reasoning-based methods that use large language models to produce rationales and verdicts. These generative approaches emphasize explanation and contextual reasoning and typically operate on existing tool outputs and benchmark assumptions.

Despite this progress, two unresolved limitations persist. First, because evaluations rely on coarse oracles, it remains unclear whether reported detections correspond to the root cause or its manifestation. Second, while generative reasoning-based triage produces natural-language explanations, their structure and content are rarely constrained in ways that support reliable downstream usage. Underreported explanations may omit the governing conditions, data flows, or triggering operations necessary to justify actionability, whereas overreported explanations may dilute the distinguishing signals that separate true positives from benign patterns, complicating both human validation and machine learning of stable decision boundaries. Moreover, prompt-based LLM assessments are inherently non-deterministic and susceptible to hallucination, raising concerns about whether plausible rationales faithfully reflect the operations underlying a warning.

These limitations create a need for an evaluation that measures detection against explicit bug and error locations to determine whether tools detect the root cause of vulnerabilities or their manifestation. On the triage side, normalizing warning semantics into structured, concise explanations that expose the relevant operations and governing conditions provides a representation over which a supervised classifier can learn to discriminate warnings aligned with the bug from incidental ones—combining LLM-based explanation generation with PTM-based classification in a way prior work has not explored. This motivates the research question of this chapter (RQ3): **To what extent can ML-based triage identify actionable SAST warnings?**

4.4 Methodology

This section describes the end to end experimental pipeline used to evaluate SAST tools and ML-based warning triage. Section 4.4.1 defines the selection criteria, configuration constraints, and execution environment for the analyzed SAST tools, ensuring build independence, default rule coverage, and SARIF interoperability under controlled conditions. Section 4.4.2 then details the construction of the training corpus derived from vulnerability fix commits, where SAST warnings are labeled through pre and post fix location matching. Section 4.4.3 introduces the reasoning layer that generates compact natural language explanations of code behavior at those warning locations and trains a supervised classifier to distinguish actionable from unactionable warnings based on those explanations. Finally, Section 4.4.4 formalizes the evaluation protocol for tool detection and assessment under bug and error oracles, and defines the metrics used to quantify both baseline and triaged performance.

4.4.1 Selection, Configuration, and Execution of SAST Tools

The tool selection process is guided by the benchmark and by the broader experimental pipeline. The aim of this work is not to extend static analysis coverage through custom rule engineering, but to evaluate whether a reasoning layer can improve the triage of findings produced by widely used SAST tools. The selection therefore starts from the analyzers identified in the related work: a literature derived list of tools commonly used in vulnerability detection studies, categorized by analysis technique, programming language support, and CWE coverage. From this set, the following criteria determine which tools are included in the pipeline:

- **Build independence**[1]**:** The analyzer must operate directly on source code without requiring compilation, build reproduction, or environment reconstruction.

  > *Motivation:* Many repositories later used to generate training data lack reproducible build systems. Build independent analysis supports rapid identification of potential bug locations, while false positive reduction is delegated to the ML-based reasoning layer.

- **Default rule coverage:** The tool must include built in rules capable of detecting the defect classes present in the benchmark, without requiring custom rules or external rulepacks.

  > *Motivation:* The objective is to assess the effect of ML-based triage, not improvements arising from hand crafted rules. Using only default rule sets reflects realistic behavior and ensures that observed gains are attributable to the pipeline rather than rule tuning.

- **Output compatibility (SARIF):** The tool must emit SARIF natively.

[1] This excludes analyzers such as `Coverity` and `Infer`, which rely on deeper integration with build systems and may achieve higher precision. Such tools cannot be applied to projects with missing build environments; therefore, conclusions are scoped to build independent analyzers.

> *Motivation:* SARIF provides a uniform structure for file paths, code ranges, metadata, rule identifiers, and CWE mappings, enabling consistent dataset construction and downstream integration with the reasoning layer.

Applying these criteria yields four analyzers: `Cppcheck`, `Flawfinder`, `Semgrep`, and `CodeQL`. All tools are executed under a controlled, uniform configuration that reflects realistic out of the box use rather than rule engineering. Each analyzer runs in a containerized environment to ensure dependency isolation and consistent execution semantics, while preserving build independence where supported. Invocation templates, container images, and tool specific parameters are fixed across experiments and documented in Appendix C, Section C.1.3. Input handling and SARIF output processing follow a standardized pipeline to ensure structural comparability of findings across tools; the operational details of this I/O handling are likewise described in Appendix C.1.3. Together, these controls ensure that differences in results stem from analyzer behavior rather than environmental or configuration variability.

4.4.2 Constructing a Root Cause Aligned Training Dataset for ML-Based Triage

Training the triage model requires both **correct** and **incorrect** SAST warnings so that a binary classifier can learn to distinguish *actionable* findings from *unactionable* ones. Obtaining such labels at scale through manual inspection is labor intensive, which motivates the use of vulnerability fix patches as a practical proxy for ground truth. A fix commit provides concrete evidence that a bug existed and was removed, while the modified code region offers a defensible approximation of the bug location. Although this chapter evaluates SAST detection at both bug location and error location, the triage task is primarily root cause oriented: a warning is actionable when it corresponds to the bug that enables the vulnerability. This focus does not exclude cases in which bug and error locations coincide, but dataset construction is driven by bug location alignment rather than their manifestation.

To obtain a sufficiently large and diverse corpus of vulnerabilities on which SAST tools can be executed, the dataset is drawn from public vulnerability databases such as the NVD, which provide CVE records with CWE metadata. While such databases often encode vulnerabilities at the level of their manifestation, the associated fix commits modify the defective code itself. As shown in Chapters 2 and 3, the bug space is diverse, whereas manifestation is concentrated in a smaller set of memory related operations. To approximate the bug space in the benchmark, CVEs are selected from manifestation related CWE categories, including CWE-125, CWE-415, CWE-416, CWE-476, and CWE-787.The criteria used to delimit the CVE corpus[2] and to select suitable fix commits are described in Appendix C.2.1. For each retained CVE, patch analysis is performed to extract a canonical bug fix location[3]; the procedures are detailed in Appendix C.2.2.

For each vulnerability instance, analysis is scoped to the affected file and performed on both the vulnerable and fixed revisions, using the same SAST configurations as in the evaluation

[2] CVEs present in the benchmark are excluded from the training corpus to prevent data leakage.

[3] Only commits with a single code relevant diff hunk are retained, to preserve a one to one mapping between CVE instance and bug location.

pipeline. Tool warnings are matched solely by location overlap and labeled as **correct warnings** in the vulnerable snapshot when they overlap the confirmed bug location. Overlapping warnings in the fixed snapshot are labeled as **incorrect warnings**, since the bug has been removed but the tool still reports a problem at that location. This pre-fix and post-fix location based labeling strategy[4] is independent of tool reported CWE or rule semantics and yields a training corpus aligned with the actionable warning triage objective; see details in Appendix C.2.3.

4.4.3 Reasoning Layer: Explanation-Driven Warning Triage

SAST tools typically report warnings through brief, rule-centric messages that assume tool-specific expertise and provide limited insight into how the flagged code behaves in context, making both human triage and learning based filtering difficult. To address this gap, this chapter introduces a natural language *reasoning layer* that operates on normalized SAST warning instances (Section C.2.3) and converts each finding into a concise explanation of observable code behavior. The layer is constrained to characterize behavior rather than correctness: it describes how the highlighted code operates and how required conditions relate to that operation, without inferring developer intent or asserting the presence of a bug beyond what is supported by the localized code context supplied to the model. These explanations provide a uniform abstraction over heterogeneous tool outputs and serve as input to a supervised triage model that distinguishes *correct* warnings, aligned with bug locations, from *incorrect* ones. In this formulation, SAST triage becomes a binary sentence classification task over normalized descriptions of code behavior, establishing an explanation driven filtering stage for actionable warning identification.

Explanation Generation. For each SAST warning, the reasoning layer generates a single sentence explanation with the structure "*FORM OPERATION* over *FOCUS OPERAND*, where the code *EVIDENCE*," grounded in a bounded code window consisting of the tool reported line range together with fixed pre and post context (Section C.3.1). A fixed prompt template separates the normalized tool message from the localized code snippet and explicitly marks the highlighted statement (Section C.3.2). The original tool message is used only as a focus hint to determine which operation or operand aspect should be described; the instruction prohibits quoting, paraphrasing, or reusing its wording and forbids vulnerability, weakness, or CWE terminology. The explanation characterizes the relationship between the operation and its required condition using one of three behavioral forms: *Proper*, *Incorrect*, or *Missing*. *Proper* denotes that the code attempts and effectively establishes the required condition in the shown context; *Incorrect* denotes that the code attempts to establish the condition but does so ineffectively or at the wrong point; and *Missing* denotes that the operation is performed without any observable attempt to establish the required condition. Instruction variants for the training data restrict *FORM* values depending on whether the warning originates from a vulnerable or fixed revision, whereas deployment permits all three forms. These categories are only operation qualifiers and are not treated as ground

[4]Because labeling is derived symmetrically from vulnerable and fixed revisions, class balance emerges from tool behavior and no rebalancing is applied.

truth labels. Each generated explanation is incorporated into the corresponding warning instance and then reintegrated into the SARIF artifact (Section C.3.3), augmenting each warning with a description of the code behavior and conditions.

Supervised Triage Learning. These augmented warning instances form the training corpus for triage, with CVE level stratified partitioning applied to prevent leakage between related bug-fix pairs (Appendix C.4.1). Triage is formulated as binary sentence classification over the generated explanations. A transformer based sentence classifier is fine-tuned on this corpus, where each training instance corresponds to a single warning and a binary label indicating whether the tool flagged a bug related location. By operating on explanations of code behavior, the classifier learns discriminative patterns such as absent constraints, ineffective checks, or properly established conditions, while abstracting away analyzer specific rule metadata. Model architecture, optimization configuration, and evaluation protocol are detailed in Appendix C.4.2. Performance is assessed using accuracy and macro averaged F1 to ensure balanced evaluation across classes. After training, the classifier functions as a filtering layer over enriched SARIF reports. For each warning containing a generated explanation, it predicts whether the warning is *correct* or *incorrect*. Warnings predicted as *correct* are retained in the triaged SARIF file, while those predicted as *incorrect* are removed. The filtering operates on SARIF `results` entries without re-executing static analysis, preserving file paths, line ranges, and rule identifiers (Appendix C.4.3).

Ablations. During development, multiple language-model backends and classifier training configurations were evaluated to determine stable explanation quality and classification behavior. Variations included alternative LLMs for explanation generation and multiple epoch settings for classifier fine-tuning. The reported configuration reflects the setting that provided the most consistent validation performance without evidence of overfitting. Details are documented in Appendix C.3 and Appendix C.4.

4.4.4 Evaluation Design: Detection and Assessment of SAST Findings

Empirical SAST evaluations commonly distinguish between two dimensions: *detection*, that is, whether a tool emits warnings that can be matched to known vulnerabilities under a location-based oracle, and *assessment*, that is, whether those warnings characterize the underlying weakness type. The present design retains this separation between detection and assessment while grounding both dimensions in exact line oracles over the benchmark from Chapter 3. Analysis is restricted to files linked to benchmark locations, thereby defining a fixed, observable search space that reflects localization and semantic relevance rather than project-wide reporting scope. Metrics are computed per tool, with the unit of analysis defined as the exact ground truth line location. Results are aggregated per tool over all benchmark instances, and bug location and error location oracles are evaluated independently. Assessment is conditioned on detection and therefore applies only to location-matched warnings.

Detection (bug vs. error localization). Following Lipp et al. [140], root cause and manifestation locations are treated as distinct evaluation targets. Each reported start line in the warning is matched independently against two exact line oracles: the bug location and the error location. A warning is considered location-matched only if its start line coincides with the corresponding ground truth line, that is, the warning line equals the bug or error line. The matching and deduplication procedures are specified in Appendix C.5.1. When multiple warnings from the same tool refer to the same file and ground truth line, they are collapsed into a single true positive. If the bug location and error location coincide for a benchmark instance, a matching warning is counted as a true positive under both oracles. Detection metrics are reported separately for each oracle to distinguish between bug and error localization.

Assessment (operation match). Assessment examines whether warnings that match a ground-truth location also report the recorded operation at that location. The *reported operation* is the BF operation [19, 18] derived from the tool-reported CWE or rule, and the *recorded operation* is the operation assigned to the benchmark location. When necessary, the rule message is consulted to disambiguate the reported operation. A location-matching warning is considered a *correct warning* if its reported operation matches the recorded operation; otherwise, it is considered an *incidental warning*. The mapping procedure and disambiguation protocol are detailed in Appendix C.5.2. Assessment results are discussed qualitatively rather than summarized through precision/recall metrics, given the small number of matched warnings.

Triage evaluation design. To evaluate the effect of ML-based triage on actionability (RQ3), detection metrics are recomputed on the triaged SARIF outputs. A warning is considered *actionable* if it matches a ground-truth location under the selected oracle; this definition accommodates benchmark instances in which bug and error locations coincide. Detection under triage is computed for both oracle locations using the same exact-line matching criteria and duplicate-collapsing rules described above. Only SARIF files that produced at least one ground-truth match in the baseline detection evaluation are considered for triage comparison, to avoid evaluating triage on files with no evaluable signal. Assessment is not recomputed for triaged outputs; triage analysis focuses on preserving defect-aligned true positives while reducing false positives.

Evaluation metrics: For each tool and each oracle, precision, recall, and F1 are computed. As in prior empirical SAST evaluations, metrics are defined over positive evidence only, since true negatives are not operationally well-defined at scale. Standard contingency counts TP, FP, and FN are computed, where TP denotes the number of unique ground-truth lines matched by at least one warning, FP denotes the number of unmatched warnings emitted within the scoped search space, and FN denotes the number of ground truth lines not recovered by any warning. The metrics are defined as $\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$, $\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}$, *and* $\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision}+\text{Recall}}$. Recall captures ground-truth coverage, precision reflects false-positive burden, and F1 summarizes their harmonic trade-off.

4.5 Results

This section reports the empirical findings of the SAST evaluation. Section 4.5.1 outlines the experimental setting and datasets. Section 4.5.2 first analyzes exact-line detection under bug and error locations. Section 4.5.3 then examines whether warnings matched at detected locations report the recorded operation. Finally, Section 4.5.4 evaluates whether explanation-driven ML triage can reduce unactionable SAST warnings while retaining warnings that match bug locations.

4.5.1 Experimental setup and datasets

This section describes the empirical setting used to evaluate both baseline and ML-based SAST triage. Evaluation is organized along three dimensions: (i) exact-line detection under bug and error locations, (ii) operation alignment of matching warnings, and (iii) the impact of explanation-driven triage on preserving warnings aligned with the bug location. The following paragraphs describe the benchmark characteristics, execution environment, training corpus for triage, and reporting conventions used to produce the results.

Evaluation dataset. The evaluation dataset is the benchmark introduced in Chapter 3. It covers **17 open-source projects—8 libraries**, **7 utilities**, and **2 servers**—and contains **30 vulnerability instances**. Due to two vulnerabilities that trigger multiple crashes and overlapping bug locations, the dataset yields **28 distinct bug locations** and **29 distinct error locations**. Across instances, root cause labels span **18 distinct CWEs**, whereas manifestation sites (reads and writes) concentrate into **5 CWE classes**, showing that manifestation labels are less diverse than bug labels. Each instance provides paired bug and error locations together with CWE and operation labels, enabling exact-line matching under both oracles. Tool execution is restricted to files containing benchmark locations, consistent with the evaluation protocol defined in Appendix C.5.1; this ensures that reported performance reflects localization within the scoped files rather than full-project reporting breadth.

Table 4.3: Warning counts within benchmark-scoped files before and after deduplication.

Tool	Bug-scoped files		Error-scoped files	
	Raw	Deduplicated	Raw	Deduplicated
CodeQL	1308	1073	775	710
Cppcheck	1401	1017	1346	984
Flawfinder	770	726	728	674
Semgrep	2377	1858	1662	1365

Tool execution and environment. All experiments are conducted in reproducible, containerized environments as described in Appendix C.1.2. Each analyzer is executed using the fixed invocation templates and rule configurations detailed in Appendix C.1.3. For `Semgrep`, the official C security and correctness rules are augmented with a curated community ruleset to compensate for

limited default coverage (Appendix C.1.3). Table 4.3 summarizes the volume of warnings generated within the scoped evaluation files before and after deduplication (Appendix C.5.1). Raw counts reflect all SARIF diagnostics emitted for files containing bug or error locations, while deduplicated counts collapse identical diagnostics and region-equivalent warnings per tool prior to metric computation.

Training dataset. The training corpus for ML-based triage is derived from CVE–patch pairs used as supervision for bug localization. The initial collection contains **807 CVE–patch pairs** across **213 repositories**, with change-size distribution:

$$n < 5 : \ 296, \quad 5 \leq n < 10 : \ 270, \quad 10 \leq n < 100 : \ 229, \quad n \geq 100 : \ 12.$$

After single-file and single-hunk filtering, the dataset retains **369 bugs** from **111 projects**, with hunk sizes distributed as follows: 1-line (148), 2-line (164), 3-lines (34), 4-lines (16), 5-lines (5), 6-lines (2). Executing the selected SAST tools on the subset of buildable projects yields **135 location-matched warnings** spanning **11 CWEs**. Among these, **63** correspond to vulnerable revisions (true positives) and **72** to fixed revisions (false positives), forming a balanced supervision corpus for training the classifier.

Reporting conventions. Detection results are reported per tool and per oracle using the contingency counts TP, FP, and FN defined in Section 4.4.4. Aggregation is performed globally per tool over all benchmark instances; metrics under the bug-location and error-location oracles are computed independently. Precision, recall, and F1 are derived from these counts as specified in Section 4.4.4, while implementation details for matching and deduplication are documented in Appendix C.5.1. Assessment based on operations follows the rule-to-operation mapping procedure described in Appendix C.5.2 and is presented qualitatively. For triage experiments, detection metrics are recomputed on triaged SARIF outputs using the same matching protocol, enabling direct comparison of true-positive preservation and false-positive reduction.

4.5.2 Detection of SAST tools under bug and error locations

This subsection compares the detection of SAST tools under two exact-line oracles: the bug location and the error location. The aim is to determine whether tool warnings align more often with the root cause or with the manifestation.

Table 4.4 reports detection under exact-line matching for both oracles. Matches remain sparse: at the bug location, tools detect between 0 and 2 instances (recall 0.0%–7.1%), while at the error location they detect between 2 and 4 instances (recall 6.9%–13.8%). Only *Semgrep* and *Flawfinder* detect any bug locations, driven by `OSV-2020-55` (both tools) and `OSV-2020-525` (Semgrep only); notably, the bug and error locations coincide for `OSV-2020-55`. At the error location, *CodeQL* and *Semgrep* achieve the highest recall (13.8%), while *CodeQL* attains the best error-location F1 (1.1%). Across all tools and both oracles, precision remains below 1%, and

Table 4.4: Detection at bug- and error-sites using exact-line matching.

Tool	**Bug-site** (|GT|=28)					**Error-site** (|GT|=29)				
	TP	FP	Prec.	Rec.	F1	TP	FP	Prec.	Rec.	F1
CodeQL	0	1073	0.0%	0.0%	0.0%	4	706	0.6%	13.8%	1.1%
Cppcheck	0	1017	0.0%	0.0%	0.0%	2	982	0.2%	6.9%	0.4%
Flawfinder	1	725	0.1%	3.6%	0.3%	3	671	0.4%	10.3%	0.9%
Semgrep	2	1856	0.1%	7.1%	0.2%	4	1361	0.3%	13.8%	0.6%

Prec.: Precision, **Rec.**: Recall; Across all tools, 2 bug and 8 error locations are detected (union of TPs);

false positives dominate the warning volume, keeping F1 below 1.1% overall. Despite the total true positives across tools (e.g., 13 at the error location), these correspond to only 2 distinct bug locations and 8 distinct error locations, reflecting substantial overlap in what different tools detect.

Finding 1. SAST tools detect error locations more often than bug locations, and do so with near-zero precision. Across tools, recall is consistently higher at error locations than at bug locations, indicating that warnings are more likely to detect the manifestation than the root cause. Bug detection is limited to two OSV cases (`OSV-2020-55` and `OSV-2020-525`), with `OSV-2020-55` exhibiting coincident bug and error locations. At the same time, extreme over-reporting persists: for every detected case, tools emit hundreds to thousands of unmatched warnings, yielding precision below 1% and F1 below 1.1%.

4.5.3 Assessment of SAST tool warnings at detected locations

Detection determines whether a warning matches a ground-truth location, but it does not indicate whether the warning corresponds to the recorded operation at that location. Assessment, therefore, compares the reported operation by the tool at the detected location with the operation recorded in the benchmark. Figures 4.1 and 4.2 summarize these mappings as Sankey diagrams, showing whether tools match or mismatch the relevant operation at the detected locations.

Bug location assessment. Figure 4.1 shows the assessment of warnings matched at bug locations. Because few warnings match bug locations, the observed tool CWEs fall into a small set whose derived operations map to BF *Data Type* (CALL, COERCE) and *Memory* (WRITE) models. The warnings matched at the bug location shared across tools (`OSV-2020-55`) are incidental in both cases: `Flawfinder` reports WRITE, while `Semgrep` reports CALL. The warning for `OSV-2020-525` is also incidental, because `Semgrep` reports COERCE. `OSV-2020-55` is especially ambiguous because its bug and error locations coincide: the matched line is a `memcpy`-style call site, but the bug is the missing length check. Bug-detecting warnings are therefore incidental because their reported operations do not match the recorded operations (*Verify*).

Error location assessment. Figure 4.2 shows the assessment of warnings matched at error locations. CWE-415 is represented with a DEREFERENCE operation; however, `Semgrep` matches

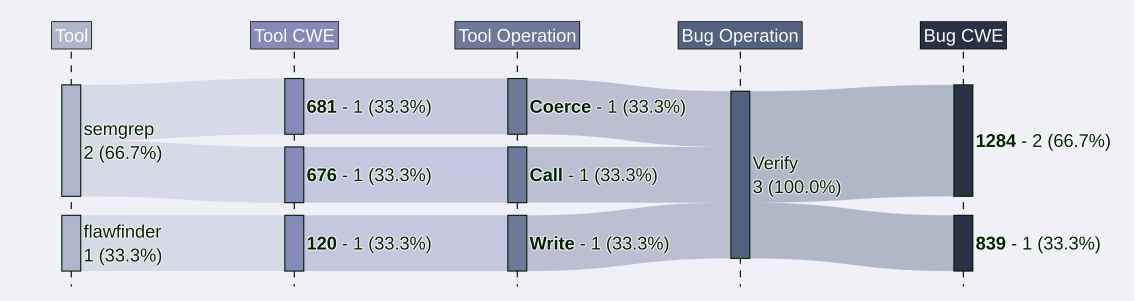


Figure 4.1: Reported CWEs and respective operations for warnings detecting bug locations.

both corresponding locations with CWE-762, which maps to DEALLOCATE, making these warnings incidental despite the correct line match. `Flawfinder` matches the WRITE operation for one entry, again `OSV-2020-55`, but reports another matched error location as READ. Beyond this correct case, tool operations largely fall into BF *Data Checks* (VERIFY, VALIDATE) and *Data Type* (DECLARE, CAST, CALL) models. The remaining warnings matched at error locations are therefore predominantly incidental: they occur at the correct line but do not match the recorded operation. A representative example is `OSV-2021-304`, where `Flawfinder` flags CWE-134 due to a format string within `snprintf`, yet the failure is driven by an invalid pointer (`&ja3_str[ja3_str_len]`) because `ja3_str_len` is not verified to remain within bounds, and the loop keeps appending data regardless of remaining capacity; the out-of-bounds memory access is represented as WRITE. `Semgrep` flags the same error line with CWE-676, which can be interpreted as a CALL, rendering the warning incidental.

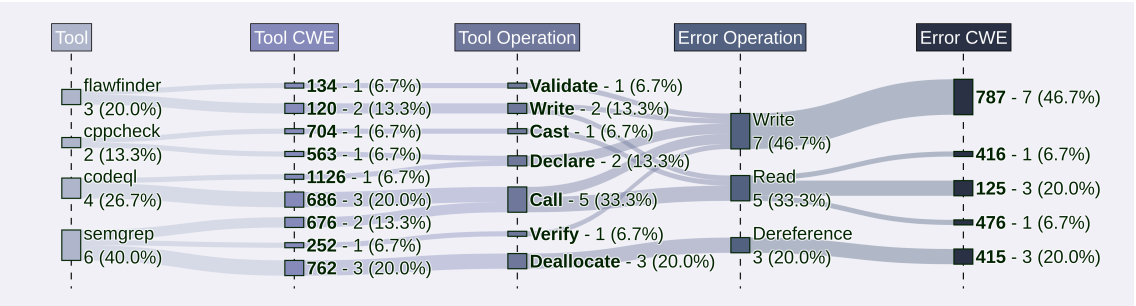


Figure 4.2: Reported CWEs and respective operations for warnings detecting error locations.

Finding 2. Location matched warnings provide limited information about assessment. Even when tools match the correct lines, the operations they report often differ from the operation recorded in the benchmark. At bug locations, tool operations concentrate in BF *Data Type* and *Memory* operations, while the bugs correspond to *Data Checks* operations. At error locations, only `Flawfinder`'s WRITE match is correct, while all remaining warnings are incidental. Cases where the bug and error locations coincide, such as *Missing* bugs, are especially ambiguous and make assessment harder.

4.5.4 Triage of SAST tool warnings at detected locations

This subsection evaluates whether ML-based triage can reduce the number of unactionable SAST warnings while retaining warnings that match bug locations. ROC analysis on the evaluation dataset, reported in Appendix C.4.4, provides a threshold-sensitive view of the classifier's ranking behavior and further clarifies the separability limits underlying the observed triage outcomes.

Table 4.5: Bug-site triage results on matched files overall and per tool. Precision/Recall/F1 are reported as percentages.

Tool	**Baseline**						**+Triage**					
	TP	FP	FN	Prec.	Rec.	F1	TP	FP	FN	Prec.	Rec.	F1
Flawfinder	1	18	0	5.3%	100.0%	10.0%	1	14	0	6.7%	100.0%	12.5%
Semgrep	2	92	0	2.1%	100.0%	4.2%	2	56	0	3.4%	100.0%	6.7%
Overall	3	110	0	2.7%	100.0%	5.2%	3	70	0	4.1%	100.0%	7.9%

Table 4.5 shows that, at bug locations, triage preserves all true positives (TP = 3, FN = 0) while removing 40 false positives, from FP = 110 to FP = 70.[5] As a result, precision improves from 2.7% to 4.1% and F1 from 5.2% to 7.9%. Per tool, the same trend holds: *Semgrep* accounts for most bug-location warnings and achieves the largest reduction in false positives (92 → 56), while *Flawfinder* improves modestly (18 → 14).

Table 4.6: Error-site triage results on matched files overall and per tool. Precision/Recall/F1 are reported as percentages.

Tool	**Baseline**						**+Triage**					
	TP	FP	FN	Prec.	Rec.	F1	TP	FP	FN	Prec.	Rec.	F1
CodeQL	4	339	0	1.2%	100.0%	2.3%	3	255	1	1.2%	75.0%	2.3%
Cppcheck	2	47	0	4.1%	100.0%	7.8%	1	33	1	2.9%	50.0%	5.6%
Flawfinder	3	212	0	1.4%	100.0%	2.8%	2	38	1	5.0%	66.7%	9.3%
Semgrep	4	275	0	1.4%	100.0%	2.8%	3	132	1	2.2%	75.0%	4.3%
Overall	13	873	0	1.5%	100.0%	2.9%	9	458	4	1.9%	69.2%	3.7%

At error locations (Table 4.6), triage removes nearly half of the false positives (873 → 458, i.e., 415 filtered), but at the cost of losing 4 true positives (13 → 9), yielding recall 69.2%. Despite the recall drop, precision increases from 1.5% to 1.9% and F1 from 2.9% to 3.7%. The per-tool breakdown shows uneven effects: *Flawfinder* benefits most in precision, from 1.4% to 5.0%, and in F1, from 2.8% to 9.3%, because many of its matched error warnings are filtered as unactionable, while *CodeQL* remains largely limited by precision even after filtering. The retained error-location warnings are concentrated in cases where the bug and error locations are close. [6]

[5] Bug-location results rely on a small number of true positives, making precision and F1 estimates sensitive to small variations. In such low-support settings, resampling-based confidence intervals would be wide, and the reported gains should be interpreted as indicative trends rather than stable estimates.

[6] Retained cases include seven same-function cases (bug-error distance of 0–3 lines) and two cross-function cases.

Finding 3. In this setting, ML-based triage can identify actionable SAST warnings by reducing unactionable warnings while preserving all warnings that match bug locations. It removes 40 false positives from 110 at bug locations, increasing precision from 2.7% to 4.1% and F1 from 5.2% to 7.9%. At error locations, it removes 415 false positives from 873, but also loses 4 true positives, reducing recall to 69.2% while increasing precision from 1.5% to 1.9% and F1 from 2.9% to 3.7%.

4.6 Discussion

This section consolidates the chapter's findings on detection, assessment, and actionability. Subsection 4.6.1 interprets the evaluation results in terms of how often tools recover the root cause instead of the manifestation, how reliably they characterize operations at those locations, and how triage affects which warnings remain actionable. Subsection 4.6.2 draws out what the results mean for empirical research, automation and tooling, and practitioners, focusing on how evaluation practices, SAST workflows, and triage design should be adapted in each case. Subsection 4.6.3 summarizes limitations on the scope of the evaluation, the training data for triage, and the statistical robustness of results, and discusses how these constraints affect external, construct, and conclusion validity. Subsection 4.6.4 outlines future work that broadens the evaluation, strengthens the learned notion of actionability, and relaxes matching criteria in a controlled way.

4.6.1 Interpretation of results

The results show that SAST tools more often locate the manifestation of a vulnerability than its root cause. Assessment shows that most location-matched warnings are incidental, because reported operations frequently differ from recorded ones, and *Missing* bugs make this harder since correct assessment requires reasoning about absent operations. Triage preserves bug-matching warnings and retains nearby error-matching warnings when bug-related code is included in the prompt window, compensating for cases where the tools miss the bug.

SAST tools tend to detect the manifestation of a vulnerability rather than its root cause. The detection results show that SAST tools match more error locations than bug locations. This tendency is further supported by the distance between bug and error locations: $9/13$ error-matching warnings occur within the same function as the bug, yet few tools match the bug location itself. The bug-matching and error-matching warning sets are also largely disjoint: the vulnerabilities matched at bug locations are generally not the same ones matched at error locations, indicating that tools tend to match one location along the vulnerability chain. For example, `Semgrep` matches the bug location of `OSV-2020-525` but not its error location a few lines later. The `OSV-2020-55` case is the only exception — the only vulnerability detected by `Flawfinder` and `Semgrep` at both bug and error locations — and only because those locations coincide, making a location-matching warning ambiguous about whether it reflects the root cause or its manifestation.

Location-matched warnings provide limited information for assessment. The assessment results show that reported operations do not reliably match the recorded operations at either location. At error locations, the single correct warning is likely incidental: `Flawfinder` reports WRITE for both `OSV-2020-55` and `OSV-2020-71`, yet only the former corresponds to a WRITE — the latter is a READ — suggesting a pattern of memory operations co-occurring at error locations rather than correct assessment. `OSV-2020-55` is also the only detected case where bug and error locations coincide, and this is characteristic of *Missing* bugs: a location match there can still reflect the manifestation rather than the root cause, since the relevant operation is absent. Since the benchmark is dominated by *Missing* bugs and all bug-matching warnings correspond to *Missing* VERIFY, this pattern explains why they are incidental. Location-matched warnings, therefore, tend to capture recognizable operations at error locations while missing the operations that make the program unsafe at bug locations.

ML-based triage can identify actionable SAST warnings. The triage results show that classifying and filtering warning explanations can reduce unactionable warnings while preserving all warnings that match bug locations. Bug-matching warnings are preserved because the classifier was trained on explanations generated for those locations and therefore directly encodes information about the bug. Some error-matching warnings are also retained, which the bug–error distances help explain: when the error is close to the bug, as in the seven same-function cases with up to three lines of distance, the prompt window in the reasoning layer still includes bug-related code even when the warning is matched at the error location. When bug and error locations are farther apart, especially across functions or files, that shared context disappears, and error-matching warnings are more likely to be filtered.

4.6.2 Implications for research, tooling, and practice

The detection, assessment, and triage results presented in this chapter carry implications for empirical research, for automation and tooling, and for practitioners. The implications are grounded in how SAST tools behave under bug and error oracles, how closely the matched warnings align with relevant operations, and how ML-based triage over explanations addresses the actionability of warnings, and inform the design and evaluation of SAST workflows across these audiences.

For empirical research. Because SAST tools more often produce error-matching warnings than bug-matching ones—and the two sets are largely disjoint—evaluations that do not distinguish between them can end up crediting tools for matching the manifestation of a vulnerability rather than its root cause. Assessment compounds this: the reported operations in location-matched warnings mostly diverge from recorded operations at both locations. For empirical work, this means that detection rates from evaluations that do not separate matches by location along the vulnerability chain may conflate root cause with manifestation, and that evaluations relying on coarse location oracles, such as function- or file-level, risk mismeasuring detection altogether. Evaluations that

additionally do not verify whether reported operations in location-matched warnings correspond to recorded ones risk counting incidental matches as genuine detection.

For automation and tooling. Because the evaluated pipeline reduces false positives while preserving bug-matching warnings, natural-language explanations of SAST findings constitute a viable representation space for learning a decision function that distinguishes actionable from unactionable warnings. For automation, this means that LLMs can articulate the semantic context at candidate locations into structured explanations that SAST tools cannot express, and that hallucination risk in the verdict can be contained at the classification step by fine-tuning a PTM-based classifier on those representations to learn a supervised decision function for actionability. These implications are most direct within the evaluated scope, where the classifier is trained on explanations generated for bug-matching warnings in CVEs and evaluated on a benchmark dominated by *Missing* bugs, and may not generalize to different bug distributions or tool configurations.

For practitioners. Because the evaluated SAST tools report many warnings while achieving near-zero precision in the benchmark, their detection is, in this setting, effectively limited to producing noisy locations near bugs, and ML-based triage can retain some of these as error-matching warnings, where retention is sensitive to the code context. For practice, this means that SAST tools are more useful as generators of candidate locations for triage than as standalone detectors, and that triage, when applied selectively to warnings in bug-prone software contexts, could surface locations that guide detection toward the bug, even when the cause is missing and the exact operation is not visible. These implications are most direct for tools that heavily overreport and for settings where the cost and latency of generating and classifying explanations remain manageable.

4.6.3 Limitations and threats to validity

This study has several limitations that should be considered when interpreting the results. These concern the scope of the benchmark and tool selection, the construction of the training corpus for the classifier, and the statistical robustness of results, which affect external, construct, and conclusion validity. Although detection, assessment, and triage are measured under a precise notion of actionability, these constraints remain relevant for interpreting how the findings may apply beyond this setting and motivate future work with broader benchmarks and training corpora.

Scope of benchmark and tool selection. The benchmark comprises 30 fuzzing-derived vulnerabilities, biasing the sample toward C/C++ memory-safety vulnerabilities in library software projects, which also constrains the tool selection to four build-independent, SARIF-emitting analyzers under default rule configurations — reflecting realistic out-of-the-box usage but excluding tools that require full builds, extensive rule engineering, or project-specific configuration. This bounds external validity in two directions: the findings may not generalize across programming languages, vulnerability classes, or discovery processes, and the results characterize lightweight

tools in default configurations rather than the full spectrum of SAST deployments. The observed patterns of detection, assessment, and triage should therefore be interpreted within this scope.

Classifier training corpus. The training corpus relies on approximating bug locations from patches, restricting bug-location extraction to fix commits with a single, small diff hunk. The labeling strategy depends on running SAST tools and obtaining bug-matching warnings in both the vulnerable and fixed snapshots; although it is independent of tool-reported CWEs or rule semantics, it may limit the corpus to the amount, distribution, and kinds of bugs the selected tools can detect. Because explanations are generated from those warnings, they can inherit tool-specific biases, where analyzers with generic messages may yield less precise explanations, while those with specific messages may produce richer descriptions. This may affect construct validity, because the learned notion of an actionable warning is grounded in the subset of bugs that are both locatable from patches and detectable by the selected tools, and the precision of that notion depends on the quality of the explanations in the training data. It may also affect internal validity, because triage gains may vary systematically across tools, favoring analyzers whose warnings carry sufficient semantic detail over those that emit generic messages.

Statistical robustness of strict matching. In the evaluation, the detection is grounded in warnings that match the exact bug and error line, which is precise but restrictive, since warnings that match surrounding locations are excluded, and other matching criteria are not evaluated. Under this strict matching, true positive counts remain low under both oracles, especially at bug locations, which further propagates into assessment and constrains triage. This affects conclusion validity, because measured recall and F1 can change materially if a single warning match is gained or lost, and the small number of true positives is insufficient to support statistically robust claims. The reported metrics should therefore be interpreted as estimates under this strict matching, rather than as bounds on what detection, assessment, and triage could achieve under more tolerant criteria.

4.6.4 Future Work

This chapter suggests several directions for future work. The main direction is to broaden the empirical basis by expanding the tool set and scaling and diversifying the benchmark and training corpora, so that detection and triage patterns can be tested beyond the current combination of tools, projects, and vulnerabilities. Another direction is to evaluate detection under graduated matching criteria in a controlled way while preserving the distinction between bug and error locations. These directions are intended to broaden the scope and robustness of the present study.

Broadening benchmark and tool set. One direction is to broaden the evaluation tool set and benchmark beyond the current four SAST tools and the corpus of 30 fuzzing-derived vulnerabilities in C/C++ projects. A natural first step is to enlarge the existing benchmark with additional vulnerabilities of the same types in the same projects, then extend it to further vulnerability classes,

additional projects, and eventually other programming languages. On the tooling side, a complementary progression is to include further lightweight SAST tools that prior empirical evaluations have underrepresented and then gradually incorporate analyzers based on sounder techniques, such as formal methods. This would address the limitation that the present results primarily characterize a few lightweight tools over a small set of memory-safety vulnerabilities, by testing whether the observed patterns in detection, assessment, and triage persist as the evaluation scope broadens.

Scaling and diversifying the training corpus. Another direction is to expand the training corpus for triage by incorporating a wider range of CVEs, vulnerability classes, and tools beyond those currently used in this chapter. This would address the limitation that the learned notion of actionability is grounded in the subset of bugs that are both locatable from compact, single-file patch changes and detectable by the selected tools, by exposing the classifier to a broader distribution of bug patterns and explanation types. It would enable more reliable calibration of the decision boundary for actionability and it would help determine whether the observed triage trends persist when the training corpus reflects greater diversity in bug structure and warning specificity.

Detection under graduated matching criteria. A further direction is to evaluate detection and triage under graduated matching criteria that extend beyond exact line matching while preserving the distinction between bug and error locations. One approach is to introduce margins around the bug location—such as matching within a bounded window of n lines before and after the canonical line—so that warnings at nearby statements can be treated as alternative notions of actionability and compared against the exact-line baseline. This would address the limitation that the current evaluation treats only exact bug-line matches which keeps true positive counts small, by explicitly measuring how often surrounding warnings would have altered detection and triage metrics. It would enable more statistically robust estimates of recall and F1 and would help characterize which proximity thresholds yield practical gains in actionability without collapsing the distinction between root cause and manifestation.

4.7 Summary

Objective. SAST tools can scan large codebases and flag code locations, but it is unclear whether detections correspond to the bug or to its manifestations, and without addressing that, it is also unclear whether any given warning is worth acting on. This chapter therefore evaluates three dimensions of tool behavior: (i) whether detections reflect *bug locations* or *error locations*; (ii) whether warnings characterize what is wrong at the matched locations; and (iii) whether retaining only warnings that characterize bugs can improve the *actionability* of SAST tool outputs.

Method. Four SAST tools (`CodeQL`, `Semgrep`, `Cppcheck`, `Flawfinder`) were evaluated on the benchmark from Chapter 3 at bug locations and error locations, using exact-line matching. Matched warnings were further assessed by comparing the operation derived from each warning's

CWE against the operation recorded in the benchmark. To evaluate actionability, a reasoning layer generates a structured explanation of code behavior at each warning location, which a PTM-based classifier uses to distinguish warnings that correspond to bug locations from those that do not. The classifier was trained on a corpus derived from CVEs in memory-safety CWE families: diff analysis of fix commits identified a single bug location per patch, and overlap with SAST warnings on both vulnerable and fixed revisions assigned correct and incorrect labels.

Findings. First, detection was limited under exact-line matching and consistently higher at error locations than at bug locations; bug-matching and error-matching warning sets were largely disjoint, suggesting that tools tend to match one location along the vulnerability chain rather than both. Second, most location-matched warnings were incidental: even when a tool matched a ground-truth line, the reported operation frequently differed from the one recorded in the benchmark; *Missing* bugs are especially difficult, since the relevant operation is absent, and coincident bug and error locations make assessment harder. Third, ML-based triage reduced unactionable warnings while preserving all warnings that match bug locations; at error locations, triage removed nearly half of the false positives but also lost some true positives, with retained cases concentrated in entries where bug and error locations are close, suggesting that proximity may preserve enough bug-related context for the classifier to identify those warnings as actionable.

Implications. These results suggest that evaluations should distinguish detection at bug locations from detection at error locations, since coarse oracles risk mismeasuring detection altogether, and evaluations that do not verify whether reported operations match recorded ones risk counting incidental matches as genuine detections. For automation, natural-language explanations of code behavior constitute a viable representation for learning to distinguish actionable from unactionable warnings, with LLM-generated explanations providing structure that SAST tools cannot express and a fine-tuned classifier containing hallucination risk at the classification step. For practice, SAST tools are more useful as generators of candidate locations for triage than as standalone detectors, and triaged error-matching warnings close to the bug location may still serve as entry points toward the root cause.

Chapter 5

Conclusions and Outlook

This chapter concludes the thesis by consolidating its contributions on vulnerability representation, benchmark ground truth, and warning actionability. It revisits the three research questions in sequence, summarizing how the thesis establishes the security bug—characterized through its root cause and software context—as the central unit of analysis, operationalizes the vulnerability representation into executable benchmark instances with explicit bug and error locations, and evaluates SAST tools and ML-based triage under oracles defined at those two reference locations and assessment criteria grounded in the root cause.

Beyond restating results, the chapter synthesizes findings that emerge only when reconstruction, reproduction, and evaluation are considered together. It highlights the effects of representing vulnerabilities primarily through their manifestations, the frequent non-equivalence of bug and error locations, and the explanatory role of *Missing* versus *Incorrect* bugs in shaping both labeling ambiguity and tool behavior. These insights motivate a coherent set of implications for vulnerability research methodology, benchmark construction, and tool design and evaluation.

The chapter closes with limitations that reflect the thesis design trade-offs—prioritizing fidelity and reproducibility over breadth—outlines future directions that extend the proposed evaluation framework, and offers concluding remarks on why ground truth that distinguishes the bug from the resulting error is necessary for meaningful evaluation and for progress toward reliable vulnerability detection and repair.

5.1 Revisiting Research Questions

This thesis evaluates vulnerability analysis techniques by distinguishing the bug from its manifestations. It represents vulnerabilities as *bug* → *error* → *failure* chains and defines ground truth through two reference locations: the *bug location*, localized from the patch, and the *error location*, localized from the crash report. This section revisits the three research questions and summarizes the main findings for each. RQ1 identifies recurring bug profiles and the error and failure outcomes through which they manifest; RQ2 examines the benchmark attributes needed to preserve the distinction between the bug and its manifestation during evaluation; and RQ3 evaluates what current SAST tools recover under that benchmark and whether ML-based triage can help retain actionable warnings.

RQ1. What are the profiles of software security bugs and their manifestations?

RQ1 asks whether the security bugs in the analyzed corpus exhibit recurring, interpretable patterns when characterized by *cause* and *operation* across software context, and how those patterns relate to the resulting errors and failures through which the bugs propagate. To answer this question, the thesis reconstructs each case as a structured *bug* → *error* → *failure* chain [1] and characterizes the bug through tuples of *cause*, *operation*, *software type*, *functional area*, and *programming language*. On that basis, the first part of the question examines recurring combinations of root cause across software context, while the second examines how those bug profiles relate to the downstream errors and failures through which vulnerabilities manifest.

The first part of the question finds that *Data Processing* components are the most frequent locus of bugs for both *Missing* and *Incorrect* causes in the curated corpus. *Missing* bugs occur most often as omitted *Data Checks*, especially in PHP web applications, whereas *Incorrect* bugs recur mainly as *Data Type* mistakes in C. These observations indicate that recurring combinations of root cause across software context can be identified within the scope of the analyzed cases.

The second part of the question finds that similar failures can arise through different error paths. In the analyzed corpus, both *Missing* and *Incorrect* bugs most often culminate in *Code Execution*; however, *Missing* cases reach this outcome through *Semantic Corruption* or *Authority Violation*, whereas *Incorrect* cases propagate through *State Corruption*. These observations are consistent with the broader argument of the thesis that evaluation should preserve the distinction between the bug and its manifestation rather than rely solely on the manifestation.

RQ2. Which vulnerability attributes are required in a benchmark for evaluation?

RQ2 asks what a vulnerability benchmark must preserve so that evaluation outcomes reflect actual detection capability rather than incidental agreement with crash context. To answer this question, the thesis constructs executable benchmark instances from fuzzing-derived vulnerabilities and records two reference locations for each case: the bug location, derived from the patch, and

[1] In the simplified form, where the *error* corresponds to the *final error* in the BF (see Section 1.2)

the error location, derived from the crash report obtained by replaying the crashing input. This design allows evaluating whether a technique identifies the bug in the code or only the runtime point at which the manifestation becomes visible.

The results indicate both structural and semantic separation between the bug and its manifestation. Bug and error locations frequently do not coincide in the reproduced vulnerabilities, and this separation can persist even within the same function. When the two locations cross function boundaries, the bug may lie along call paths ranging from a single edge to deeper call chains, indicating that the crash report can localize the manifestation far from the underlying bug. A similar separation appears in the operations recorded at these locations: at the error location, operations in these 30 crash reproductions are concentrated in a small set of program actions tied to memory usage, while the operations associated with the bug reveal substantially broader diversity. In effect, many distinct bug operations can converge on the same limited set of error site operations.

These findings indicate that a reliable evaluation benchmark should preserve (i) explicit recording of both the bug location and the error location whenever they can be reconstructed, and (ii) the improper operation derived from patch artifacts, so that evaluation does not collapse distinct bugs into a single notion of correctness defined only by the manifestation.

RQ3. To what extent can ML-based triage identify actionable SAST warnings?

RQ3 examines what current SAST tools recover when evaluation distinguishes the bug location from the error location, and whether ML-based triage can improve the actionability of reported warnings. To answer this question, the thesis evaluates SAST tools on the benchmark introduced in Chapter 3 under the two exact line oracles and assesses whether matched warnings convey meaningful information about the bug by comparing operations derived from the CWE labels reported by the tools with the bug and error operations recorded in the benchmark. It then introduces an ML-based triage layer that learns from short natural-language explanations derived from the warning message and the localized code context to filter unactionable findings.

The evaluation results indicate that recall tends to be higher at the error location than at the bug location, suggesting that the analyzed tools align more often with the manifestation than with the underlying bug. Because the tools over-report, the precision is very low, leaving many warnings unactionable and limiting their practical usefulness. Beyond localization, matched warnings often provide only limited information about the bug: even when tools report the correct lines, their operations derived from the reported CWE often differ from the bug and error operations in the benchmark. This indicates that a location match alone does not mean that the warning identifies the bug under analysis, especially when the bug and error locations are close or coincide.

The final part of RQ3 finds that the ML-based triage layer can filter a substantial share of false positives while preserving most of the warnings that match the bug location, but not those that match the error location. In the evaluated setting, this suggests that the triage step can retain the warnings considered as actionable, namely those that align with the bug location and therefore point to code whose repair would remove the vulnerability. These findings remain specific to the evaluated benchmark, tool set, and triage scope.

5.2 Synthesis of Key Findings

This section brings together findings that become clearer when Chapters 2–4 are considered jointly. Across the thesis, vulnerabilities are examined as *bug* → *error* → *failure* chains, reconstructed from patch artifacts, reported outcomes, and runtime evidence. This representation reveals recurring patterns and structure, highlights the prevalence of *Missing* bugs and their relation to the error, and shows why distinguishing the bug from the error matters for evaluation and triage. When that structure is not preserved, distinct bugs can be reduced to similar manifestations, making it harder to interpret what the data represent and what tools actually detect.

Vulnerability structure across profiling and benchmarking

Chapters 2 and 3 indicate that the vulnerabilities analyzed in this thesis exhibit a structure that becomes visible only when the bug is related to its manifestation, and that software context provides an additional lens through which it becomes possible to explain why bugs arise and how they manifest depending on the implementation setting in which they occur.

Chapter 2 introduces the *bug* → *error* → *failure* chain as the representation of a vulnerability and uses it to analyze how bugs relate to their manifestations across software context. While the bug is reconstructed from patch artifacts, the manifestation side of the chain is reconstructed from the CVE description, so part of the analysis characterizes propagation patterns based on reported outcomes. It nevertheless reveals that vulnerabilities similar at the level of failure can arise through different bug → error pathways, and that those pathways vary with software context, especially programming language: *State Corruption* occurs more often in C/C++ cases, while *Semantic Corruption* or *Authority Violation* appear more often in PHP Web Applications.

To ground in code the vulnerability representation introduced in Chapter 2, Chapter 3 reproduces vulnerabilities in C/C++ and measures the separation between the bug and the error locations, adding a positional dimension to that structure. In the reproduced cases, bug and error locations frequently do not coincide: separation persists even within the same function, and when locations cross function boundaries, call-graph distances range from a single call edge to deep call chains. This extends the results of Chapter 2: vulnerability structure includes not only how bugs propagate into manifestations, but also how the positional relation between the bug and the error varies across reproduced cases. The next subsection shows that variation is itself related to the semantics of the bug.

Missing data check bugs as a dominant pattern

Across both the curated corpus (Chapter 2) and the reproduced benchmark instances (Chapter 3), roughly 70% of reconstructed vulnerabilities are characterized by a *Missing* cause rather than an *Incorrect* cause. This recurring pattern helps explain why the bug is often harder to characterize and localize than its manifestation: when the relevant condition is absent from the code, what becomes visible is the point at which the program first operates on an improper operand.

Chapter 2 makes this pattern visible through bug profiles. In that corpus, *Missing* bugs account for 72.7% of the cases and concentrate in *Sanitize*, *Validate*, and especially *Verify*. These operations reflect expected conditions on data that should have been established before the program proceeds. Such bugs are harder to localize as explicit program actions, since the problem is that a required check or constraint was omitted rather than that an existing operation was performed incorrectly. By contrast, *Incorrect* bugs are less frequent and more dispersed across operations. At this stage, the analysis suggests that many vulnerabilities arise because expected properties of input, such as bounds, validity, or consistency, were assumed but never established.

Chapter 3 shows that the same pattern persists in the reproduced C/C++ cases, where 66.7% of bugs are also characterized by a *Missing* cause. Bug operations remain more diverse, but they again concentrate in *Verify*, mostly under a *Missing* cause; *Missing* also appears in *Clear*, *Reposition*, and *Initialize*. By contrast, *Incorrect* bugs are less frequent and remain distributed across a wider range of operations, consistent with Chapter 2. When the distance between bug and error is measured, a further difference appears: for bugs with a *Missing* cause, the observed error more often remains within the same function as the bug, whereas *Incorrect* bugs more often propagate across functions and files. This indicates that bug–error distance is also shaped by the root cause. That distinction becomes important in the next subsection, where SAST tools are examined to determine whether they match the bug or only its manifestation.

Effects of distinguishing bug and error in evaluation

Chapter 4 shows that evaluation depends strongly on which reference location is treated as correct. Distinguishing the bug location from the error location changes both how tool performance is measured and how warning actionability is assessed. Triage can improve actionability, but only when the code context around the flagged location retains enough information about the bug.

Under that evaluation, SAST tools more often align with the observed error than with the underlying bug, but with very low precision. A location match alone is not sufficient to conclude that a warning identifies the bug: even when warnings match the reference lines, the operations derived from the reported CWEs often differ from those recorded in the benchmark, suggesting that many matched warnings are incidental. Read together with Chapters 2 and 3, this shows that the bug can be harder to identify than its manifestation, which limits what evaluation can measure and what tools can reliably detect. This is especially relevant in a corpus where many bugs are characterized by *Missing* causes and where bug and error do not always coincide.

The ML-based triage results complement this picture. In the evaluated setting, the triage layer preserves warnings that match the bug location while filtering some that match only the error location, consistent with the thesis notion of actionability. One plausible explanation is that the classifier is trained on explanations derived from bug-matching warnings, so the signal it learns is more closely tied to bug context than to error context. This makes the positional findings of Chapter 3 relevant to triage: when bug and error are close, the code context around an error-matching warning may still retain enough information about the bug to support an actionable explanation; when they are farther apart, that signal may no longer be present. These results

suggest that ML-based triage can reduce false positives while preserving relevant warnings, even at error locations, when the available context still contains enough information about the bug.

5.3 Implications

Vulnerability understanding and evaluation become more reliable when vulnerabilities are represented as *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chains. Datasets and benchmarks must support reconstruction and reproducibility while preserving the distinction between bug and error; this representation enables analysis of vulnerability patterns and requires evaluation methods that distinguish what is being detected; and, in practice, tool outputs must identify the bug and relate findings across the chain to support actionable repair, with triage benefiting from operating over these structures.

Implications for datasets and benchmarks

A central implication from Chapter 2 and Chapter 3 is that datasets and benchmarks must support two linked requirements: vulnerabilities must be reconstructable from public records and reproducible as executable instances. For reconstruction, datasets must provide enough structured information to recover the bug and its manifestation from reported artifacts, while for reproducibility, benchmarks must preserve the software and execution artifacts needed to replay and validate those cases over time.

For reconstruction, current vulnerability records often lack the structured information needed to support reliable automation and large-scale analysis. In practice, key metadata needed to identify the relevant software and trace it to public source code is often incomplete, inconsistent, or missing, which complicates the recovery of patches and supporting artifacts. CVE descriptions present a similar problem: many contain grammatical errors, follow no consistent template, or omit details about the root cause, consequences, and impact needed for reconstruction. Without such improvements, vulnerability datasets remain limited in scale and utility.

For reproducibility, benchmarks require more than a description, metadata, and a patch. In Chapter 3, benchmark construction links OSV records to NVD CVEs to identify vulnerabilities that can be reproduced for concrete manifestation evidence at runtime. To support buildability, replayability, and fix verification, these records must link to vulnerable project versions, reproducer inputs, crash reports, and fixing commits that remain available and usable. These requirements are often undermined by ecosystem drift and underspecified metadata, which make software harder to rebuild and validate over time. Benchmarks should therefore package the software and execution artifacts needed to reproduce each case.

Underlying both conditions is a question of representation: datasets and benchmarks must be explicit about which elements of the vulnerability their ground truth captures. Chapter 3 shows that operations at the error location are concentrated and can arise from many distinct bugs, so treating the manifestation as ground truth without clarifying its role in the chain leaves downstream tasks without a reliable oracle. This ambiguity follows from reporting practices examined in Chapter 2, where vulnerabilities are described primarily through their manifestation, collapsing the *bug* $\rightarrow$

error → *failure* chain into a single label. Datasets and benchmarks should therefore treat the bug and the error as distinct elements of the vulnerability.

Implications for research

A central implication is that analyzing vulnerabilities as *bug* → *error* → *failure* chains across software context reveals patterns that are not visible when considering manifestation alone. Chapter 2 shows that, when reconstructed in this way, vulnerabilities can be organized into bug profiles that explain where bugs concentrate and how they propagate into errors and failures. This also reveals that similar failures can arise from different bugs, and that propagation depends on the surrounding software context. These results suggest that such patterns should be studied at larger scale to determine whether they remain stable, can be consolidated, or whether additional patterns emerge.

Closely related is the need to distinguish explicitly between elements of the vulnerability chain during evaluation. Chapter 3 shows that the bug location and the error location frequently diverge. This implies that empirical studies should state explicitly which element of the chain their oracle targets and, where feasible, report results separately for each location. Without this distinction, evaluation may misrepresent what is being detected, for example by counting matches at the error location as if they identified the bug, which can lead to overestimating detection. Chapter 4 shows that this occurs in practice, with SAST tools matching error locations more often than bug locations. Even when warnings match the reference lines, they may describe operations that differ from those recorded in the oracle, making such matches incidental. This indicates that, for a warning to be meaningful, it must be evaluated both by its location and by whether it captures the operation that is actually improper at that location.

Implications for tools and practice

The main implication for practitioners is that, as observed in Chapter 2, data-processing components, such as parsers and upload handlers, emerge as recurring vulnerable locations in software, meaning these hotspots should be prioritized. Because the reconstruction is grounded in fixing patches, these implications correspond to concrete repair actions: *Missing* bugs benefit most from validation, verification, and sanitization, whereas *Incorrect* bugs require correctness in arithmetic, memory management, and control flow.

The main implication for tools is that, since developers fix the bug to remove the vulnerability, warnings must identify those locations to be considered actionable. As shown in Chapter 2 and Chapter 3, many cases involve **Missing** cause, where the bug is not an incorrect statement but omitted logic that may not have a single explicit location in code. In such cases, automation must infer that a relevant program action is absent, rather than merely flag code that appears wrong. This suggests that reporting warnings as independent findings is insufficient for actionability, and that representations capturing vulnerability chains in code and at runtime can provide a more informative basis for inspection and repair.

Triage can also benefit from operating over vulnerability chains rather than isolated warnings. In the evaluated setting of Chapter 4, triage operates on natural-language explanations derived from warning messages and limited source-code context, and results show that warnings matching error locations are more likely to be retained when that context also captures information about the bug, which typically occurs when the two locations are close. When the bug and error locations are further apart, this shared context is less likely to be present, making relevant warnings harder to retain. This suggests that triage would benefit from representations that explicitly connect these locations, particularly when bug–error separation spans functions or files.

5.4 Limitations and Threats to Validity

The limitations of this thesis concern two recurring threats, corpus scope and labeling reliability, that also affect the evaluation and its outcomes. Chapters 2 and 3 were curated independently, so their external validity is bounded separately, but both rest on small, heavily filtered corpora and share the same construct validity threat in root-cause labeling, which propagates into Chapter 4 alongside low detection rates that constrain conclusion validity. Both corpora exhibit a similar distribution of bug causes, partially mitigating concerns about construct validity. These constraints motivate future directions on broader corpora and evaluation settings. The following paragraphs address the primary and secondary limitations of each chapter in turn.

Chapter 2 corpus scope, representativeness, and reliability. Of the 4,324 Critical-severity CVEs initially considered, successive filtering for reporting quality, entity extraction, and patch traceability reduced the corpus to 66 cases. This bounds external validity: the observed bug-cause distributions should be interpreted as corpus-conditional rather than representative of the broader vulnerability population, and likely skew toward better-documented, more cleanly attributable cases. A secondary limitation is patch noise, which concerns construct validity, because it can prevent confident attribution of the bug to a specific patch region; complex cases were excluded rather than labeled with reduced confidence. A related limitation concerns patches that perform multiple actions, which also affect construct validity because root cause labeling requires interpretive judgment, compounded when the operation falls outside the current BF's coverage and must be mapped to the closest CWE weakness; ambiguous cases were resolved through multiple iterations of AI-assisted analysis. Mitigation of construct validity threats also involved documented disambiguation heuristics, consistency checks, and validation against issue reports and fix commit messages, but no inter-rater agreement check was performed.

Chapter 3 corpus selection, scope, and stability. Of the 1,520 OSV entries with product and CWE overlap in NVD, only 69 were reference-matched, and automated crash reproduction was feasible for 30. This bounds external validity in a direction distinct from Chapter 2's, because the OSS-Fuzz origin of the corpus biases it toward memory-safety vulnerabilities in C/C++ programs and toward components amenable to fuzzing, such as libraries; as a result, the reproduced

set should be read as characterizing what can be reproduced from such artifacts rather than the broader vulnerability population. A secondary concern is construct validity: this chapter reuses Chapter 2's cause and operation labeling schema, so the same labeling ambiguities and the inter-rater agreement gap discussed above recur here. A further concern is the reproducibility of the corpus over time since reproduction depends on OSS-Fuzz infrastructure, repositories, build dependencies, and toolchains that may change or disappear, and although containerizing the software and execution artifacts needed for each case mitigates this fragility, it does not remove the dependence of the corpus on artifacts outside the control of the thesis.

Chapter 4 scope, selection, and robustness. The evaluation inherits the scope constraints described above for Chapter 3; tool selection is further limited to four build-independent, SARIF-emitting analyzers under default configurations. This bounds external validity in two directions: the findings may not generalize across languages, vulnerability classes, or discovery processes, and the results characterize lightweight tools in default configurations rather than the full spectrum of SAST deployments. A secondary limitation is what the classifier learns as actionable, which concerns construct validity, because the training corpus reflects the subset of bugs that are both locatable from small patches and detectable by the selected tools. Further, the measured recall and F1 metrics, especially at bug locations, are limited by low true positive counts, which are insufficient to support statistically robust claims and thus concern conclusion validity; the reported metrics should therefore be read as estimates under strict exact-line matching rather than as bounds on what detection, assessment, or triage could achieve under more tolerant criteria.

5.5 Future Directions

This thesis suggests several directions for future work that extend its scope and address its primary constraints. Scaling the reconstruction and labeling workflow through AI-based assistance would reduce corpus attrition and improve reliability across all three empirical stages. Extending the reproduction pipeline and evaluation to additional projects, vulnerability classes, and tool types would test whether the observed patterns generalize beyond the current empirical setting. The following paragraphs elaborate on these directions for each chapter.

Chapter 2 scalability, corpus coverage, and schema refinement. The primary direction is scaling the reconstruction and labeling workflow through AI-based assistance. Large language models could compensate for incomplete or irregular CVE records by normalizing descriptions, recovering missing details, and improving traceability from products to repositories, while agentic workflows could support patch scanning, bug localization, and structured explanation generation. Together, these would reduce the corpus attrition identified in the limitations and shift the human role toward reviewing AI-generated cases rather than constructing them from scratch; as the dataset grows, the manually curated cases could also serve as a benchmark for evaluating how

closely AI-assisted methods approximate expert judgment. Secondary directions include expanding coverage beyond Critical-severity CVEs to test whether the observed bug-cause profiles remain stable across severity levels, and refining software context attributes, such as functional area, to improve profile separability without over-fragmenting the schema.

Chapter 3 coverage, labeling automation, and pipeline scalability. The primary direction is extending the reproduction pipeline to additional programming languages, project types, and vulnerability classes by building on OSS-Fuzz's multi-language support and adapting the Docker-based build and sanitizer configurations accordingly. This would reduce the selection bias toward C/C++ memory-safety vulnerabilities identified in the limitations and broaden the benchmark's generalizability. The AI-assisted labeling and inter-rater validation direction discussed above for Chapter 2 applies here as well, since the root cause labels in this chapter carry the same construct validity threat; the manually labeled cases from both chapters could serve as few-shot context or fine-tuning data for that classification task. A further direction specific to this chapter is improving OSV–CVE linkage and root-cause localization: incorporating affected product versions as a fallback matching criterion could increase corpus coverage beyond the current reference-overlap dependency, and fault localization guided by fuzzing could reduce the manual effort required to identify the bug location from the crashing input.

Chapter 4 evaluation scope, training diversity, and matching criteria. The primary direction is broadening the evaluation beyond the current four lightweight SAST tools and 30 fuzzing-derived vulnerabilities. A natural progression is to build on the corpus extension direction discussed above for Chapter 3, and then to incorporate analyzers based on sounder techniques, such as formal methods, to test whether the observed detection, assessment, and triage patterns persist as the scope broadens. A secondary direction is expanding the training corpus for the triage classifier to reflect a wider range of bug patterns and explanation types beyond those in the current study; this relates to the AI-assisted labeling direction discussed above for Chapters 2 and 3, since a broader and more reliably labeled corpus would also improve the training signal. A further direction is to evaluate detection and triage under graduated matching criteria within a bounded line window while preserving the distinction between bugs and errors; this would yield more robust recall and F1 estimates and help identify proximity thresholds for practical gains in actionability.

5.6 Concluding Remarks

Vulnerability analysis and evaluation depend on reliable ground truth about what a vulnerability is — yet public records, standards, and benchmarks have historically described vulnerabilities through their manifestations, conflating root cause with observable consequence. This thesis addressed that gap by operationalizing vulnerabilities as *bug* $\rightarrow$ *error* $\rightarrow$ *failure* chains grounded in open-source artifacts, and by constructing, from that representation, a reproducible benchmark that records both the bug and error locations as distinct elements of ground truth. Building on

this foundation, it then evaluated SAST tools and ML-based triage under oracles defined at both reference locations, using root-cause characterizations to assess not only where warnings appear but what they indicate about the bug.

The central finding is that vulnerabilities have a recoverable structure — recurring bug profiles can be identified in the analyzed corpus, and bugs propagate through distinct error paths depending on whether the underlying cause is *Missing* or *Incorrect*, even when they converge on the same failure. This structure is also positional: bug and error locations frequently diverge across functions and files, and the operations recorded at each location differ substantially, with error operations concentrating in a small set of memory actions while bug operations span a broader range. Across both reference locations, SAST tool warnings align more often with the error location than the bug location, and even location-matched warnings often report operations that differ from those recorded in the benchmark. ML-based triage can preserve warnings that match the bug location while reducing false positives, but its effectiveness for error-matching warnings narrows as the distance between bug and error locations increases, since the code context around the warning carries less information about the bug. These findings indicate that evaluating detection and triage without distinguishing the bug from its manifestation underdetermines what tools identify.

These findings matter because vulnerability analysis and tool evaluation have largely treated manifestation labels as adequate proxies for the underlying bug, without distinguishing what a tool detects from what it reports. The results show that this conflation has measurable consequences: evaluations that do not separate matches by location along the vulnerability chain credit tools for aligning with the manifestation rather than the root cause; and evaluations that do not verify whether reported operations match recorded ones count incidental location matches as genuine detections. For *Missing* bugs in particular — where the relevant operation is absent from code — the bug location is harder to localize and the warning harder to verify, since there is no statement to flag. Distinguishing bug locations from error locations, and root cause characterizations from manifestation labels, provides a more precise basis for measuring what tools actually report and for identifying which warnings are worth acting on.

These results hold within specific, bounded settings: small corpora filtered for reconstruction quality in two independently curated studies, a reproduced benchmark of 30 fuzzing-derived C/C++ memory-safety vulnerabilities, and four lightweight SAST tools evaluated under exact-line matching in default configurations. Whether the observed bug profiles, location separation patterns, and triage behavior generalize to other languages, vulnerability classes, discovery processes, or tool types remains open. This scope reflects a deliberate prioritization of fidelity over breadth: establishing reliable ground truth required controlling for the factors most directly affecting what can be reconstructed, reproduced, and validated from public artifacts, and that constraint necessarily limits what the results can claim about the broader vulnerability population or the full spectrum of SAST deployments.

The immediate next steps — scaling the workflow, broadening the benchmark, and extending the evaluation — are elaborated in the preceding section. What those directions collectively serve is a question the thesis reframed but did not resolve: whether detection tools can identify repair-

relevant locations when the relevant operations are absent from the vulnerable code. Answering that question at a broader scale, and eventually connecting detection and triage to repair strategies that target the most frequent bug causes, is the natural continuation of the work begun here.

References

[1] Seyed Ali Akhavani, Behzad Ousat, and Amin Kharraz. “Open Source, Open Threats? Investigating Security Challenges in Open-Source Software”. In: *ArXiv* abs/2506.12995 (2025). URL: https://api.semanticscholar.org/CorpusID:279403206.

[2] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. “Empirical analysis of security vulnerabilities in Python packages”. In: *Empirical Software Engineering* 28 (2021), pp. 1–37. URL: https://api.semanticscholar.org/CorpusID:232255898.

[3] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. “On the Discoverability of npm Vulnerabilities in Node.js Projects”. In: *ACM Transactions on Software Engineering and Methodology* 32 (2022), pp. 1–27.

[4] Henrique Alves, Baldoino Fonseca dos Santos Neto, and Nuno Antunes. “Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study”. In: *2016 12th European Dependable Computing Conference (EDCC)* (2016), pp. 37–44. URL: https://api.semanticscholar.org/CorpusID:5591135.

[5] Richard Amankwah, Patrick Kwaku Kudjo, and Samuel Yeboah Antwi. “Evaluation of Software Vulnerability Detection Methods and Tools: A Review”. In: *International Journal of Computer Applications* 169 (2017), pp. 22–27.

[6] Afsah Anwar, Ahmed A. Abusnaina, Songqing Chen, Frank H. Li, and David A. Mohaisen. “Cleaning the NVD: Comprehensive Quality Assessment, Improvements, and Analyses”. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)* (2020), pp. 1–2.

[7] Masaki Aota, Hideaki Kanehara, Masaki Kubo, Noboru Murata, Bo Sun, and Takeshi Takahashi. “Automation of Vulnerability Classification from its Description using Machine Learning”. In: *2020 IEEE Symposium on Computers and Communications (ISCC)* (2020), pp. 1–7.

[8] Andrei Arusoaie, Stefan Ciobaca, Vlad Constantin Craciun, Dragos Gavrilut, and Dorel Lucanu. “A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code”. In: *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (2017), pp. 161–168.

[9] Kulkarni Ashish and Jagannath Aghav. “Automated techniques and tools for program analysis: Survey”. In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)* (2013), pp. 1–7.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[11] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. "A Survey of Symbolic Execution Techniques". In: *ACM Computing Surveys (CSUR)* 51 (2016), pp. 1–39.

[12] Anil Bazaz and James D. Arthur. "Towards a Taxonomy of Vulnerabilities". In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)* (2007), 163a–163a.

[13] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software". In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering* (2021).

[14] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. "SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), pp. 1059–1070.

[15] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Reza Abbasi, Joel Cameron Frank, Simon Wörner, and Thorsten Holz. "AURORA: Statistical Crash Analysis for Automated Root Cause Explanation". In: *USENIX Security Symposium*. 2020. URL: https://api.semanticscholar.org/CorpusID:215805353.

[16] Irena Bojanova. *Bugs Framework (BF): Formalizing Cybersecurity Weaknesses and Vulnerabilities*. NIST Special Publication 800-231. Gaithersburg, MD: National Institute of Standards and Technology, July 2024. DOI: 10.6028/NIST.SP.800-231. URL: https://csrc.nist.gov/pubs/sp/800/231/final.

[17] Irena Bojanova. "Comprehensively Labeled Weakness and Vulnerability Datasets via Unambiguous Formal Bugs Framework Specifications". In: *IT Professional* 26.1 (2024), pp. 60–68. DOI: 10.1109/MITP.2024.3358970.

[18] Irena Bojanova. *NIST Bugs Framework (BF)*. https://pages.nist.gov/BF/. Accessed: 2026-06-15. 2026.

[19] Irena Bojanova, Paul E. Black, Yaacov Yesha, and Yan Wu. "The Bugs Framework (BF): A Structured Approach to Express Bugs". In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2016, pp. 175–182. DOI: 10.1109/QRS.2016.29.

[20] Irena Bojanova, Carlos Eduardo C. Galhardo, and Irena Bojanova. "Heartbleed Revisited: Is it Just a Buffer Over-Read?" In: *IT Professional* 25 (2023), pp. 83–89. URL: https://api.semanticscholar.org/CorpusID:258642021.

[21] Irena Bojanova, John Guerrerio, and Eduard Pinconschi. *Labeling Software Security Vulnerabilities*. Presentation (ITL Science Day), National Institute of Standards and Technology (NIST). Accessed 2026-01-14. Oct. 2023. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=957025.

[22] Teresa Nicole Brooks. "Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning Systems". In: *ArXiv* abs/1702.06162 (2017).

[23] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models are Few-Shot Learners". In: *ArXiv* abs/2005.14165 (2020).

[24] Q. C. Bui, R. Scandariato, and N. E. Díaz Ferreyra. "Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques". In: *MSR*. 2022.

[25] Robert Byers, David Waltermire, and Christopher Turner. "Collaborative Vulnerability Metadata Acceptance Process (CVMAP) for CVE Numbering Authorities (CNAs) and Authorized Data Publishers". In: 2020.

[26] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *NASA Formal Methods*. 2011.

[27] Andrea Capiluppi, Nemitari Ajienka, Nour Ali, Mahir Arzoky, Steve Counsell, Giuseppe Destefanis, Alina Dana Miron, Bhaveet Nagaria, Rumyana Neykova, Martin J. Shepperd, Stephen Swift, and Allan Tucker. "Using the Lexicon from Source Code to Determine Application Domain". In: *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering* (2020). URL: https://api.semanticscholar.org/CorpusID:213384251.

[28] Sebastián Castro. *Off-by-one overflow explained*. Accessed 21-October-2020. Oct. 2016. URL: https://csl.com.co/en/off-by-one-explained/.

[29] Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. "Transformer-based vulnerability detection in code at EditTime: Zero-shot, few-shot, or fine-tuning?" In: *arXiv preprint arXiv:2306.01754* (2023).

[30] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. "An Empirical Study of Static Analysis Tools for Secure Code Review". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2024). URL: https://api.semanticscholar.org/CorpusID:271244450.

[31] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. "Hawkeye: Towards a Desired Directed Grey-box Fuzzer". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018). URL: https://api.semanticscholar.org/CorpusID:53109507.

[32] Tianyu Chen, Lin Li, Bingjie Shan, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. "Identifying Vulnerable Third-Party Java Libraries from Textual Descriptions of Vulnerabilities and Libraries". In: 2023. URL: https://api.semanticscholar.org/CorpusID:259937077.

[33] Brian Chess. *Assessing Application Vulnerabilities: A 360 Degree Approach*. Tech. rep. 2215 Bridgepointe Pkwy. San Mateo, California 94404: Fortify Inc., Oct. 2011, p. 9. URL: http://www.seankirk.com/Downloads/Fortify_360_Whitepaper.pdf.

[34] Steve Christey. *PLOVER - Preliminary List Of Vulnerability Examples for Researchers*. Tech. rep. Version 0.24. Draft document. MITRE Corporation, Mar. 2006. URL: https://cwe.mitre.org/documents/sources/PLOVER.pdf.

[35] MITRE Corporation. *About CWE*. Accessed 3-June-2022. June 2022. URL: https://cwe.mitre.org/about/index.html.

[36] MITRE Corporation. *CVE CNAs*. Accessed 26-June-2023. June 2023. URL: https://www.cve.org/ProgramOrganization/CNAs.

[37] MITRE Corporation. *CVE Glossary*. Accessed 7-June-2022. June 2022. URL: https://www.cve.org/ResourcesSupport/Glossary#glossaryVulnerability.

[38] MITRE Corporation. *CVE Overview*. Accessed 26-June-2023. June 2023. URL: https://www.cve.org/About/Overview.

[39] MITRE Corporation. *CWE Glossary*. Accessed 8-June-2022. June 2022. URL: https://cwe.mitre.org/documents/glossary/index.html.

[40] MITRE Corporation. *CWE List*. Accessed 31-May-2025. June 2025. URL: https://cwe.mitre.org/data/index.html.

[41] Roland Croft, Muhammad Ali Babar, and Mehdi Kholoosi. "Data Quality for Software Vulnerability Datasets". In: *ArXiv* abs/2301.05456 (2023).

[42] Roland Croft, Muhammad Ali Babar, and Li Li. "An Investigation into Inconsistency of Software Vulnerability Severity across Data Sources". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2021), pp. 338–348.

[43] Roland Croft, Dominic Newlands, Ziyu Chen, and Muhammad Ali Babar. "An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing". In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2021).

[44] Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27 (2008), pp. 1165–1178.

[45] Tobias Dam and Sebastian Neumaier. "Towards Measuring Vulnerabilities and Exposures in Open-Source Packages". In: 2022. URL: https://api.semanticscholar.org/CorpusID:258564931.

[46] Wenjing Dang, Kaixuan Li, Sen Chen, Zhenwei Zhuo, Lyu Bing Zhang, and Zheli Liu. "Real-World Usability of Vulnerability Proof-of-Concepts: A Comprehensive Study". In: *ArXiv* abs/2510.18448 (2025).

[47] Siddhartha Shankar Das, Edoardo Serra, Mahantesh Halappanavar, Alex Pothen, and Ehab Al-Shaer. "V2W-BERT: A Framework for Effective Hierarchical Multiclass Classification of Software Vulnerabilities". In: *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)* (2021), pp. 1–12.

[48] Steven M. Christey David E. Mann. *Towards a Common Enumeration of Vulnerabilities*. Tech. rep. 202 Burlington Rd., Bedford MA 01730: The MITRE Corporation, Jan. 1999, p. 13. URL: https://www.cve.org/Resources/General/Towards-a-Common-Enumeration-of-Vulnerabilities.pdf.

[49] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *North American Chapter of the Association for Computational Linguistics*. 2019. URL: https://api.semanticscholar.org/CorpusID:52967399.

[50] Gabriel Díaz and Juan Ramón Bermejo. "Static analysis of source code security: Assessment of tools against SAMATE tests". In: *Inf. Softw. Technol.* 55 (2013), pp. 1462–1476.

[51] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 110–121. URL: https://api.semanticscholar.org/CorpusID:206579056.

[52] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. "Towards the Detection of Inconsistencies in Public Security Vulnerability Reports". In: *USENIX Security Symposium*. 2019.

[53] Hao Du, Chengshan Gao, and Zhaotun Ma. "Test Case Generation Techniques for Fault Root Cause Localization". In: *Proceedings of the 3rd International Conference on Signal Processing, Computer Networks and Communications* (2024). URL: https://api.semanticscholar.org/CorpusID:276470877.

[54] Richard A. Dubniczky, Krisztofer Zolt'an Horv'at, Tamás Bisztray, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Norbert Tihanyi. "CASTLE: Benchmarking Dataset for Static Code Analyzers and LLMs towards CWE Detection". In: *Theoretical Aspects of Software Engineering*. 2025. URL: https://api.semanticscholar.org/CorpusID:276937971.

[55] Michael D. Ernst. "Static and dynamic analysis: synergy and duality". In: 2003.

[56] David Evans and David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis". In: *IEEE Softw.* 19 (2002), pp. 42–51.

[57] Onyeka Ezenwoye and Yi Liu. "Web Application Weakness Ontology Based on Vulnerability Data". In: *ArXiv* abs/2209.08067 (2022).

[58] Onyeka Ezenwoye, Yi Liu, and W. George Patten. "Classifying Common Security Vulnerabilities by Software Type". In: *SEKE*. 2020.

[59] Onyeka Ezenwoye and Eduard Pinconschi. "A Study of Fine-Tuned Language Models in Vulnerability Classification". In: *2024 12th International Symposium on Digital Forensics and Security (ISDFS)*. 2024, pp. 1–6. DOI: 10.1109/ISDFS60797.2024.10527294.

[60] Onyeka Ezenwoye, Eduard Pinconschi, and Earl H. Roberts. "Exploring AI for Vulnerability Detection and Repair". In: *2024 Cyber Awareness and Research Symposium (CARS)* (2024), pp. 1–9.

[61] Jiahao Fan, Yi Li, Shaohua Wang, and Tien Nhut Nguyen. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". In: *Proceedings of the 17th International Conference on Mining Software Repositories* (2020).

[62] Ansgar Fehnker, Mark Bradley, Ralf Huuck, and Paul A. Steckler. "Goanna static analysis at the NIST static analysis tool exposition". In: 2011.

[63] Ansgar Fehnker, Ralf Huuck, PA Jayet, Michel Lussenburg, and Felix Rauch. "Goanna - A Static Model Checker". In: *FMICS/PDMC*. 2006.

[64] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *ArXiv* abs/2002.08155 (2020). URL: https://api.semanticscholar.org/CorpusID:211171605.

[65] Andrew Feutrill, Dinesha Ranathunga, Yuval Yarom, and Matthew Roughan. "The Effect of Common Vulnerability Scoring System Metrics on Vulnerability Exploit Delay". In: *2018 Sixth International Symposium on Computing and Networking (CANDAR)* (2018), pp. 1–10.

[66] *FFmpeg Security*. https://ffmpeg.org/security.html. Accessed: 2025-12-21. 2019.

[67] Gwyn Fisher. *The Klocwork Insight™ Analysis Engine: Generating Accurate, Scalable Whole Program Analysis*. CTO. White Paper, 2009. URL: https://docplayer.net/19883491-The-klocwork-insight-analysis-engine.html.

[68] Andrew Forward and Timothy C. Lethbridge. "A taxonomy of software types to facilitate search and evidence-based software engineering". In: *Conference of the Centre for Advanced Studies on Collaborative Research*. 2008.

[69] Shivi Garg, Rajesh Kumar Singh, and Amar Kumar Mohapatra. "Analysis of software vulnerability classification based on different technical parameters". In: *Information Security Journal: A Global Perspective* 28 (2019), pp. 1–19.

[70] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. "Automatic Software Repair: A Survey". In: *IEEE Transactions on Software Engineering* 45 (2019), pp. 34–67.

[71] Xiuting Ge, Chunrong Fang, Xuanye Li, Weisong Sun, Daoyuan Wu, Juan Zhai, Shangwei Lin, Zhihong Zhao, Yang Liu, and Zhenyu Chen. "Machine Learning for Actionable Warning Identification: A Comprehensive Survey". In: *ACM Computing Surveys* 57 (2023), pp. 1–35. URL: https://api.semanticscholar.org/CorpusID:265551683.

[72] Xiuting Ge, Chunrong Fang, Quanjun Zhang, Daoyuan Wu, Bo-Chen Yu, Qirui Zheng, An Guo, Shangwei Lin, Zhihong Zhao, Yang Liu, and Zhenyu Chen. "Pre-trained Model-based Actionable Warning Identification: A Feasibility Study". In: *ACM Transactions on Software Engineering and Methodology* (2024).

[73] Christoph Gentsch. "Evaluation of Open Source Static Analysis Security Testing (SAST) Tools for C". In: 2020. URL: https://api.semanticscholar.org/CorpusID:221700166.

[74] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. "Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques". In: *ACM Computing Surveys (CSUR)* 50 (2017), pp. 1–36.

[75] GitHub. *GitHub Advisory Database*. 2022. URL: https://github.com/advisories.

[76] Antonios Gkortzis, Daniel Feitosa, and Diomidis D. Spinellis. "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities". In: *J. Syst. Softw.* 172 (2021), p. 110653.

[77] *GNU Binutils*. https://www.gnu.org/software/binutils/. Accessed: 2025-12-21. 1998.

[78] Betul Gokkaya, Leonardo Aniello, and Basel Halak. "Software supply chain: review of attacks, risk assessment strategies and security controls". In: *ArXiv* abs/2305.14157 (2023). URL: https://api.semanticscholar.org/CorpusID:258841711.

[79] Xiujun Gong, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Zhuobing Han. "Joint Prediction of Multiple Vulnerability Characteristics Through Multi-Task Learning". In: *2019 24th International Conference on Engineering of Complex Computer Systems* (2019), pp. 31–40.

[80] Google. *Open Source Vulnerability Database*. https://osv.dev/. [Online; accessed 25-May-2022]. 2022.

[81] Google. *OSS-Fuzz-Vulns*. https://github.com/google/oss-fuzz-vulns. GitHub repository. Source of disclosed OSS-Fuzz vulnerabilities for OSV. 2025.

[82] Google Open Source Security Team. *OSS-Fuzz: Continuous Fuzzing for Open Source Software*. https://github.com/google/oss-fuzz#trophies. Accessed: 2025-11-26. 2016.

[83] Anjana Gosain and Ganga Sharma. "Static Analysis: A Survey of Techniques and Tools". In: 2015.

[84] Katerina Goseva-Popstojanova and Andrei Perhinschi. "On the capability of static code analysis to detect security vulnerabilities". In: *Inf. Softw. Technol.* 68 (2015), pp. 18–33. URL: https://api.semanticscholar.org/CorpusID:19956818.

[85] Katerina Goseva-Popstojanova and Jacob Tyo. "Security Vulnerability Profiles of Mission Critical Software: Empirical Analysis of Security Related Bug Reports". In: 2017. URL: https://api.semanticscholar.org/CorpusID:955555.

[86] C. L. Goues, N.Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W.Weimer. "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs". In: *IEEE TSE* (2015).

[87] GrammaTech. *Eliminating Vulnerabilities in Third-Party Code with Binary Analysis*. Tech. rep. 2014. URL: https://cdn2.hubspot.net/hubfs/582328/PDF/third-party-code.pdf?t=1516135417449.

[88] Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Jiamou Sun. "Detecting and Augmenting Missing Key Aspects in Vulnerability Descriptions". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (2022), pp. 1–27.

[89] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. "Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities". In: *IEEE Transactions on Software Engineering* 49 (2023), pp. 5154–5188. URL: https://api.semanticscholar.org/CorpusID:264989991.

[90] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. "Learning to Predict Severity of Software Vulnerability Using Only Vulnerability Description". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), pp. 125–136.

[91] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. "Finding patterns in static analysis alerts: improving actionable alert ranking". In: *IEEE Working Conference on Mining Software Repositories*. 2014. URL: https://api.semanticscholar.org/CorpusID:1232246.

[92] Hazim Hanif, Mohd Hairul Nizam Bin Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches". In: *J. Netw. Comput. Appl.* 179 (2021), p. 103009.

[93] Nima Shiri Harzevili, Jiho Shin, Junjie Wang, and Song Wang. "Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries". In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)* (2022), pp. 27–38. URL: https://api.semanticscholar.org/CorpusID:247446763.

[94] Keno Hassler, Philipp Görz, Stephan Lipp, Thorsten Holz, and Marcel Böhme. "A Comparative Study of Fuzzers and Static Analysis Tools for Finding Memory Unsafety in C and C++". In: *ArXiv* abs/2505.22052 (2025). URL: https://api.semanticscholar.org/CorpusID:278960054.

[95] Ahmad Hazimeh, Adrián Herrera, and Mathias Payer. "Magma". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4 (2020), pp. 1–29. URL: https://api.semanticscholar.org/CorpusID:221137226.

[96] Sarah Smith Heckman and Laurie A. Williams. "A measurement framework of alert characteristics for false positive mitigation models". In: 2008.

[97] Sarah Smith Heckman and Laurie A. Williams. "A Model Building Process for Identifying Actionable Static Analysis Alerts". In: *2009 International Conference on Software Testing Verification and Validation* (2009), pp. 161–170. URL: https://api.semanticscholar.org/CorpusID:6031344.

[98] Sarah Smith Heckman and Laurie A. Williams. "A systematic literature review of actionable alert identification techniques for automated static code analysis". In: *Inf. Softw. Technol.* 53 (2011), pp. 363–387. URL: https://api.semanticscholar.org/CorpusID:7656069.

[99] Hannes Holm and Khalid Khan Afridi. "An expert-based investigation of the Common Vulnerability Scoring System". In: *Comput. Secur.* 53 (2015), pp. 18–30.

[100] Gerard J. Holzmann and Murray Hill. "UNO: Static Source Code Checking for User-Defined Properties 1". In: 2002.

[101] Qiao Huang, Xin Xia, and D. Lo. "Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), pp. 159–170. URL: https://api.semanticscholar.org/CorpusID:10552608.

[102] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. "The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study". In: *IEEE Transactions on Software Engineering* 49 (2023), pp. 44–63. URL: https://api.semanticscholar.org/CorpusID:247448700.

[103] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[104] GARTNER PEER INSIGHTS. *Checkmarx CxSAST Forges Ahead in Accuracy Benchmarks*. Tech. rep. 2020. URL: https://checkmarx.com/resources/whitepapers/cxsast-accuracy-whitepaper-1.

[105] Mohsen Iranmanesh, Sina Moradi Sabet, Sina Marefat, Ali Javidi Ghasr, Allison Wilson, Iman Sharafaldin, and Mohammad A. Tayebi. "ZeroFalse: Improving Precision in Static Analysis with LLMs". In: *ArXiv* abs/2510.02534 (2025). URL: https://api.semanticscholar.org/CorpusID:281829484.

[106] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[107] Kedrian James, Kevin Valakuzhy, Kevin Z. Snow, and Fabian Monrose. "CrashTalk: Automated Generation of Precise, Human Readable, Descriptions of Software Security Bugs". In: *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy* (2024). URL: https://api.semanticscholar.org/CorpusID:270380714.

[108] Tiantian Ji, Yue Wu, Chang Wang, Xi Zhang, and Zhongru Wang. "The Coming Era of AlphaHacking?: A Survey of Automatic Software Vulnerability Detection, Exploitation and Patching Techniques". In: *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. 2018, pp. 53–60. DOI: 10.1109/DSC.2018.00017.

[109] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. "The symptoms, causes, and repairs of bugs inside a deep learning library". In: *J. Syst. Softw.* 177 (2021), p. 110935. URL: https://api.semanticscholar.org/CorpusID:233329485.

[110] Heju Jiang, Jasvir Nagra, and Parvez Ahammad. "SoK: Applying Machine Learning in Security - A Survey". In: *ArXiv* abs/1611.03186 (2016).

[111] Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. "[Engineering Paper] Enabling the Continuous Analysis of Security Vulnerabilities with VulData7". In: *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2018, pp. 56–61. DOI: 10.1109/SCAM.2018.00014.

[112] Pontus Johnson, Robert Lagerström, Mathias Ekstedt, and Ulrik Franke. "Can the Common Vulnerability Scoring System be Trusted? A Bayesian Analysis". In: *IEEE Transactions on Dependable and Secure Computing* 15 (2018), pp. 1002–1015.

[113] Hong Jin Kang, Khai Loong Aw, and D. Lo. "Detecting False Alarms from Automatic Static Analysis Tools: How Far are We?" In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), pp. 698–709. URL: https://api.semanticscholar.org/CorpusID:246823411.

[114] Arvinder Kaur and Ruchika Nayyar. "A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code". In: *Procedia Computer Science* 171 (2020), pp. 2023–2029. URL: https://api.semanticscholar.org/CorpusID:226455565.

[115] Brandon N. Keller, Benjamin S. Meyers, and Andrew Meneely. "What Happens When We Fuzz? Investigating OSS-Fuzz Bug History". In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 2023, pp. 207–217. DOI: 10.1109/MSR59073.2023.00038.

[116] Mete Keltek, Rong Hu, Mohammadreza Fani Sani, and Ziyue Li. "LSAST: Enhancing Cybersecurity Through LLM-Supported Static Application Security Testing". In: *IFIP International Information Security Conference*. 2025.

[117] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin B. Clement, and Neel Sundaresan. "Learning to Reduce False Positives in Analytic Bug Detectors". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), pp. 1307–1316. URL: https://api.semanticscholar.org/CorpusID:247595369.

[118] Igor Khokhlov, Ahmet Emin Okutan, Ryan Bryla, Steven Simmons, and Mehdi Mirakhorli. "Automated Extraction of Software Names from Vulnerability Reports using LSTM and Expert System". In: *2022 IEEE 29th Annual Software Technology Conference (STC)* (2022).

[119] Amir A. Khwaja, Muniba Murtaza, and Hafiz Farooq Ahmed. "A security feature framework for programming languages to minimize application layer vulnerabilities". In: *Security and Privacy* 3 (2020).

[120] Sunghun Kim and Michael D. Ernst. "Which warnings should I fix first?" In: *ESEC-FSE '07*. 2007. URL: https://api.semanticscholar.org/CorpusID:9484699.

[121] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Aspects of Computing* 27 (2015).

[122] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael W. Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018). URL: https://api.semanticscholar.org/CorpusID:52127357.

[123] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. "Learning a classifier for false positive error reports emitted by static code analysis tools". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (2017). URL: https://api.semanticscholar.org/CorpusID:495409.

[124] Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. "An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)* (2019), pp. 288–299. URL: https://api.semanticscholar.org/CorpusID:174820417.

[125] Philip D. . Kuehn, Markus Bayer, Marc Wendelborn, and Christian A. Reuter. "OVANA: An Approach to Analyze and Improve the Information Quality of Vulnerability Databases". In: *Proceedings of the 16th International Conference on Availability, Reliability and Security* (2021).

[126] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. "A Taxonomy of Computer Program Security Flaws". In: 26.3 (Sept. 1994), pp. 211–254. ISSN: 0360-0300. DOI: 10.1145/185403.185412. URL: https://doi.org/10.1145/185403.185412.

[127] J. C. Laprie. "Dependability: Basic Concepts and Terminology". In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Ed. by J. C. Laprie. Vienna: Springer Vienna, 1992, pp. 3–245. ISBN: 978-3-7091-9170-5. DOI: 10.1007/978-3-7091-9170-5_1. URL: https://doi.org/10.1007/978-3-7091-9170-5_1.

[128] Triet Huynh Minh Le, Huaming Chen, and Muhammad Ali Babar. "A Survey on Data-driven Software Vulnerability Assessment and Prioritization". In: *ACM Computing Surveys* 55 (2021), pp. 1–39.

[129] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. "Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)* (2019), pp. 391–401. URL: https://api.semanticscholar.org/CorpusID:174820310.

[130] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach". In: *Proceedings of the ACM on Programming Languages* 8 (2024), pp. 474–499. URL: https://api.semanticscholar.org/CorpusID:268375318.

[131] Hongwei Li, Wenmeng Zhang, and Yongjun Wang. "Software Vulnerability Root Cause Analysis Method Based on Data Analysis". In: *2025 6th International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)* (2025), pp. 365–368. URL: https://api.semanticscholar.org/CorpusID:281894782.

[132] Kai-Jing Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. "PatchFinder: A Two-Phase Approach to Security Patch Tracing for Disclosed Vulnerabilities in Open-Source Software". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2024). URL: https://api.semanticscholar.org/CorpusID:271404420.

[133] Xiangwei Li, Xiaoning Ren, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. "Prediction of Vulnerability Characteristics Based on Vulnerability Description and Prompt Learning". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2023), pp. 604–615.

[134] Z. Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *ArXiv* abs/1801.01681 (2018).

[135] Ziyang Li, Saikat Dutta, and Mayur Naik. "LLM-Assisted Static Analysis for Detecting Security Vulnerabilities". In: *ArXiv* abs/2405.17238 (2024). URL: https://api.semanticscholar.org/CorpusID:270063578.

[136] 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". In: *IEEE Transactions on Dependable and Secure Computing* 21 (2024), pp. 4600–4618. URL: https://api.semanticscholar.org/CorpusID:267108463.

[137] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. "Vulnerabilities and Security Patches Detection in OSS: A Survey". In: *ACM Computing Surveys* 57 (2024), pp. 1–37. URL: https://api.semanticscholar.org/CorpusID:272589281.

[138] Shiyin Lin. "LLM-Driven Adaptive Source–CSink Identification and False Positive Mitigation for Static Analysis". In: *Proceedings of the 2025 8th International Conference on Computer Information Science and Artificial Intelligence* (2025). URL: https://api.semanticscholar.org/CorpusID:282812833.

[139] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. "A Survey of Transformers". In: *AI Open* 3 (2021), pp. 111–132. URL: https://api.semanticscholar.org/CorpusID:235368340.

[140] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).

[141] Bing Chang Liu, Liang Shi, Zhuhua Cai, and Min Li. "Software Vulnerability Discovery Techniques: A Survey". In: *2012 Fourth International Conference on Multimedia Information Networking and Security* (2012), pp. 152–156.

[142] Han Liu, Jian Zhang, Cen Zhang, Xiaohan Zhang, Kaixuan Li, Sen Chen, Shang-wei Lin, Yixiang Chen, Xinhua Li, and Yang Liu. "FineWAVE: Fine-Grained Warning Verification of Bugs for Automated Static Analysis Tools". In: *ArXiv* abs/2403.16032 (2024). URL: https://api.semanticscholar.org/CorpusID:268680478.

[143] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *ArXiv* abs/1907.11692 (2019). URL: https://api.semanticscholar.org/CorpusID:198953378.

[144] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html. Accessed: 2026-02-17.

[145] Ivan P. Malashin. "Generation of Natural-Language Explanations for Static-Analysis Warnings Using Single- and Multi-Objective Optimization". In: *Comput.* 14 (2025), p. 534. URL: https://api.semanticscholar.org/CorpusID:283622909.

[146] Cláudia Mamede, José Campos, Claire Le Goues, and Rui Abreu. "Interpretable Vulnerability Detection Reports". In: *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2025), pp. 2021–2032. URL: https://api.semanticscholar.org/CorpusID:285108371.

[147] Cláudia Mamede, Eduard Pinconschi, and Rui Abreu. "A transformer-based IDE plugin for vulnerability detection". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022).

[148] Cláudia Mamede, Eduard Pinconschi, Rui Abreu, and José Campos. "Exploring Transformers for Multi-Label Classification of Java Vulnerabilities". In: *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)* (2022), pp. 43–52. URL: https://api.semanticscholar.org/CorpusID:257661401.

[149] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* 47 (2018), pp. 2312–2331.

[150] Nikolai Mansourov and Djenana Campara. *System Assurance: Beyond Detecting Vulnerabilities*. First. Morgan Kaufmann, 2010. ISBN: 9780123814159.

[151] Daniel Marjamäki. *Cppcheck Design*. https://sourceforge.net/projects/cppcheck/files/Articles/cppcheck-design.pdf/download. [Online; accessed 23-December-2022]. 2014.

[152] Jeferson J.F. Martinez and Javier M. Durán. "Software Supply Chain Attacks, a Threat to Global Cybersecurity: SolarWinds' Case Study". In: *International Journal of Safety and Security Engineering* (2021). URL: https://api.semanticscholar.org/CorpusID:244056951.

[153] Matias Martinez, Laurence Duchien, and Monperrus Martin. "Automatically Extracting Instances of Code Change Patterns with AST Analysis". In: *2013 IEEE International Conference on Software Maintenance* (2013), pp. 388–391.

[154] Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Abdelouahab Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doup'e, Hammond Pearce, and Brendan Dolan-Gavitt. "ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software". In: *ArXiv* abs/2408.02153 (2024). URL: https://api.semanticscholar.org/CorpusID:271709574.

[155] Peter Mell, Karen Scarfone, and Sasha Romanosky. "Common Vulnerability Scoring System". In: *IEEE Security & Privacy* 4 (2006).

[156] Mend. *Vulnerability Database*. 2022. URL: https://www.mend.io/vulnerability-database/.

[157] Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury. “Program Environment Fuzzing”. In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (2024). URL: https://api.semanticscholar.org/CorpusID:269293860.

[158] Microsoft. *Microsoft Threat Modeling Tool threats*. https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats. Last updated: 2022-08-25. Accessed: 2026-01-13.

[159] MITRE. *Common Attack Pattern Enumeration and Classification (CAPEC)*. https://capec.mitre.org/index.html. Accessed: 2026-01-13.

[160] MITRE. *CWE - Schema Documentation - Schema Version 7.3*. https://cwe.mitre.org/documents/schema/. Accessed: 2026-01-03.

[161] MITRE and Jonathan Evans. *Key Details Phrasing*. Accessed 26-June-2023. 2020. URL: http://cveproject.github.io/docs/content/key-details-phrasing.pdf.

[162] MITRE Corporation. *CVE → CWE "Root Cause Mapping" Guidance*. https://cwe.mitre.org/documents/cwe_usage/guidance.html. Guidance for mapping CVE Records to CWE root causes; recommends Base/Variant-level CWEs and provides mapping labels and methodologies. 2025. (Visited on 09/02/2025).

[163] Lloyd Montgomery, Clara Marie Luders, and W. Maalej. “Mining Issue Trackers: Concepts and Techniques”. In: *ArXiv* abs/2403.05716 (2024).

[164] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. “.QL: Object-Oriented Queries Made Easy”. In: *Generative and Transformational Techniques in Software Engineering*. 2007.

[165] P. Jason Morrison, Kim Herzig, Brendan Murphy, and Laurie Ann Williams. “Challenges with applying vulnerability prediction models”. In: *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security* (2015).

[166] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. “Using complexity metrics to improve software security”. In: *Computer Fraud & Security* 2013 (2013), pp. 8–17.

[167] Mozilla. *Mozilla Foundation Security Advisories*. 2022. URL: https://www.mozilla.org/en-US/security/advisories/.

[168] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and G. Wang. “Understanding the Reproducibility of Crowd-reported Security Vulnerabilities”. In: *USENIX Security Symposium*. 2018. URL: https://api.semanticscholar.org/CorpusID:49317374.

[169] Amy Munson, Juanita Gomez, and Alvaro A. Cárdenas. “♪ With a Little Help from My (LLM) Friends: Enhancing Static Analysis with LLMs to Detect Software Vulnerabilities”. In: *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)* (2025), pp. 25–32. URL: https://api.semanticscholar.org/CorpusID:279324062.

[170] National Institute of Standards and Technology (NIST). *The NIST Vulntology Specification*. https://pages.nist.gov/vulntology/specification/. Accessed: 2026-01-13.

[171] Huu Hung Nguyen, Duc-Manh Tran, Yiran Cheng, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Lwin Khin Shar, Ouh Eng Lieh, Ting Zhang, and David Lo. "Mapping NVD Records to Their VFCs: How Hard is it?" In: *ArXiv* abs/2506.09702 (2025). URL: https://api.semanticscholar.org/CorpusID:279305590.

[172] Viet Hung Nguyen and Fabio Massacci. "The (un)reliability of NVD vulnerable versions data: an empirical experiment on Google Chrome vulnerabilities". In: *ArXiv* abs/1302.4133 (2013).

[173] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. "Analyzing bug fix for automatic bug cause classification". In: *J. Syst. Softw.* 163 (2020), p. 110538. URL: https://api.semanticscholar.org/CorpusID:211829736.

[174] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. "Principles of Program Analysis". In: *Springer Berlin Heidelberg*. 1999.

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

[176] NIST. *Source Code Security Analyzers*. https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers. [Online; accessed 13-December-2022]. 2022.

[177] Ahmet Emin Okutan, Peter Mell, Mehdi Mirakhorli, Igor Khokhlov, Joanna C. S. Santos, Danielle Gonzalez, and Steven Simmons. "Empirical Validation of Automated Vulnerability Curation and Characterization". In: *IEEE Transactions on Software Engineering* 49 (2023), pp. 3241–3260.

[178] Ahmet Emin Okutan and Mehdi Mirakhorli. "Predicting the Severity and Exploitability of Vulnerability Reports using Convolutional Neural Nets". In: *2022 IEEE/ACM 3rd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)* (2022), pp. 1–8.

[179] OWASP. *Source Code Analysis Tools*. https://owasp.org/www-community/Source_Code_Analysis_Tools. [Online; accessed 13-December-2022]. 2022.

[180] Sinno Jialin Pan and Qiang Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22 (2010), pp. 1345–1359. URL: https://api.semanticscholar.org/CorpusID:740063.

[181] Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. "Predicting Vulnerable Software Components through N-Gram Analysis and Statistical Feature Selection". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)* (2015), pp. 543–548.

[182] Yulei Pang, Xiaozhen Xue, and Huaying Wang. "Predicting Vulnerable Software Components through Deep Neural Network". In: *International Conference on Deep Learning Technologies*. 2017.

[183] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. "Benzene: A Practical Root Cause Analysis System with an Under-Constrained State Mutation". In: *2024 IEEE Symposium on Security and Privacy (SP)* (2024), pp. 1865–1883. URL: https://api.semanticscholar.org/CorpusID:266935209.

[184] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. "Vulnerable open source dependencies: counting those that matter". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2018). URL: https://api.semanticscholar.org/CorpusID:52123163.

[185] Alexander I. Pechenkin and Roman Demidov. "Applying Deep Learning and Vector Representation for Software Vulnerabilities Detection". In: *Proceedings of the 11th International Conference on Security of Information and Networks* (2018).

[186] José D'Abruzzo Pereira and Marco Paulo Amorim Vieira. "On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects". In: *2020 16th European Dependable Computing Conference (EDCC)* (2020), pp. 97–102.

[187] Henning Perl, Sergej Dechand, Matthew Smith, Dan Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Gülsüm Acar. "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[188] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. "Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction". In: *IEEE International Conference on Tools with Artificial Intelligence*. 2017.

[189] Frank Piessens. "A taxonomy of causes of software vulnerabilities in Internet software". In: 2002. URL: https://www2.cs.kuleuven.be/publicaties/rapporten/cw/CW346.pdf.

[190] Eduard Pinconschi, Quang-Cuong Bui, Rui Abreu, Pedro Adão, and Riccardo Scandariato. "Maestro: a platform for benchmarking automatic program repair tools on software vulnerabilities". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022). URL: https://api.semanticscholar.org/CorpusID:250562496.

[191] Eduard Pinconschi, Divya Gopinath, Rui Abreu, and Corina S. Păsăreanu. "Evaluating Deep Neural Networks in Deployment: A Comparative Study (Replicability Study)". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2024). URL: https://api.semanticscholar.org/CorpusID:271097589.

[192] Eduard Pinconschi, Sofia Reis, Chi Zhang, Rui Abreu, Hakan Erdogmus, Corina S. Păsăreanu, and Limin Jia. "Tenet: A Flexible Framework for Machine-Learning-based Vulnerability Detection". In: *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 2023, pp. 102–103. DOI: 10.1109/CAIN58948.2023.00026.

[193] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. "A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (2019), pp. 383–387.

[194] Zhuoyun Qian, Fangtian Zhong, Qin Hu, Yili Jiang, Jiaqi Huang, Mengfei Ren, and Jiguo Yu. "Software Vulnerability Analysis Across Programming Language and Program Representation Landscapes: A Survey". In: *ArXiv* abs/2503.20244 (2025). URL: https://api.semanticscholar.org/CorpusID:277321710.

[195] Tanya Brewer Robert Byers Chris Turner. *National Vulnerability Database*. 2022. URL: https://doi.org/10.18434/M3436.

[196] Fayozbek Rustamov, Juhwan Kim, Jihyeon Yu, Hyunwook Kim, and Joobeom Yun. "BugMiner: Mining the Hard-to-Reach Software Vulnerabilities through the Target-Oriented Hybrid Fuzzer". In: *Electronics* (2020). URL: https://api.semanticscholar.org/CorpusID:234439017.

[197] Abhiroop Saha and Prof. Raghavendra S G Prasad. "A Survey on Coverity Scan Analysis". In: *International Journal of Advanced Research in Computer and Communication Engineering* 11.7 (2022), pp. 345–352. DOI: 10.17148/IJARCCE.2022.11764.

[198] Luis Alberto Benthin Sanguino and Rafael Uetz. "Software Vulnerability Analysis Using CPE and CVE". In: *ArXiv* abs/1705.05347 (2017).

[199] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *ArXiv* abs/1910.01108 (2019). URL: https://api.semanticscholar.org/CorpusID:203626972.

[200] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. "Predicting Vulnerable Software Components via Text Mining". In: *IEEE Transactions on Software Engineering* 40 (2014), pp. 993–1006.

[201] SCANOSS. *purl2cpe: PURL to CPE Relationship mapping project*. GitHub repository. Commit 4fa870f, MIT License. 2025. URL: https://github.com/scanoss/purl2cpe (visited on 11/10/2025).

[202] Kostya Serebryany. "OSS-Fuzz - Google's continuous fuzzing service for open source software". In: 2017. URL: https://api.semanticscholar.org/CorpusID:69626811.

[203] Hossain Shahriar and Mohammad Zulkernine. "Mitigating program security vulnerabilities: Approaches and challenges". In: *ACM Comput. Surv.* 44 (2012), 11:1–11:46.

[204] Weijie Shao, Yuyang Gao, Fu Song, Sen Chen, and Lingling Fan. "A Comprehensive Empirical Study of Bugs in Open-Source Federated Learning Frameworks". In: 2023. URL: https://api.semanticscholar.org/CorpusID:260735562.

[205] Lwin Khin Shar and Hee Beng Kuan Tan. "Predicting common web application vulnerabilities from input validation and sanitization code patterns". In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (2012), pp. 310–313.

[206] Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. "Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing". In: *ArXiv* (2017). URL: https://api.semanticscholar.org/CorpusID:20207825.

[207] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. "Localizing Vulnerabilities Statistically From One Exploit". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (2021).

[208] Yonghee Shin, Andrew Meneely, Laurie Ann Williams, and Jason A. Osborne. "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities". In: *IEEE Transactions on Software Engineering* 37 (2011), pp. 772–787.

[209] Yonghee Shin and Laurie Ann Williams. "An initial study on the use of execution complexity metrics as indicators of software vulnerabilities". In: *SESS '11*. 2011.

[210] Shin Hwei Tan, Jooyong Yi, Yulis, S. Mechtaev, and A. Roychoudhury. "Codeflaws: a programming competition benchmark for evaluating automated program repair tools". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 180–182. DOI: 10.1109/ICSE-C.2017.76.

[211] Shin'ichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. "Test suites for benchmarks of static analysis tools". In: *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2015), pp. 12–15.

[212] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)* (2016), pp. 138–157.

[213] Snyk. *Snyk Vulnerability Database*. Snyk Documentation. Accessed 2025-11-13. 2025. URL: https://docs.snyk.io/scan-with-snyk/snyk-open-source/manage-vulnerabilities/snyk-vulnerability-database.

[214] Victor Sobreira, Thomas Durieux, Fernanda Madeiral Delfim, Monperrus Martin, and Marcelo de Almeida Maia. "Dissection of a bug dataset: Anatomy of 395 patches from Defects4J". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), pp. 130–140.

[215] Gimpel Software. *Reference Manual for PC-lint/FlexeLint: A Diagnostic Facility for C and C++*. URL: https://docplayer.net/63457731-Pc-lint-flexelint-c-and-c.html. (accessed: 15.12.2022).

[216] Marc Solé, Victor Muntés-Mulero, Annie Ibrahim Rana, and Giovani Estrada. "Survey on Models and Techniques for Root-Cause Analysis". In: *ArXiv* abs/1701.08546 (2017). URL: https://api.semanticscholar.org/CorpusID:9935621.

[217] Tim Sonnekalb, Thomas S. Heinze, and Patrick Mäder. "Deep security analysis of program code". In: *Empirical Software Engineering* 27 (1 Jan. 2022), p. 2. ISSN: 1382-3256. DOI: 10.1007/s10664-021-10029-x.

[218] National Institute of Standards and Technology. *NIST Glossary*. Accessed 7-June-2022. June 2022. URL: https://csrc.nist.gov/glossary.

[219] National Institute of Standards and Technology. *SARD benchmarks*. Accessed 17-June-2022. June 2022. URL: https://samate.nist.gov/SARD/.

[220] Clang Team. *Clang Static Analyzer Design Documentation*. https://clang.llvm.org/docs/ClangStaticAnalyzer.html. [Online; accessed 23-December-2022]. 2022.

[221] Kawouther Thabet, Jihen Bennaceur, Salma Hamza, Raoudha Ben Jemaa, and Wissem Zouaghi. "Securing the Software Supply Chain: A New Taxonomy for Attack Classification". In: *2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (2023), pp. 1–8. URL: https://api.semanticscholar.org/CorpusID:268741931.

[222] Saad Ullah, Praneeth Balasubramanian, Wenbo Guo, Amanda Burnett, Hammond Pearce, Christopher Kruegel, Giovanni Vigna, and Gianluca Stringhini. "From CVE Entries to Verifiable Exploits: An Automated Multi-Agent Framework for Reproducing CVEs". In: *ArXiv* abs/2509.01835 (2025).

[223] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need". In: *NIPS*. 2017.

[224] Veracode. *Veracode Vulnerability Database*. 2022. URL: https://sca.veracode.com/vulnerability-database/search.

[225] Andreas Wagner and Johannes Sametinger. “Using the Juliet Test Suite to compare static security scanners”. In: *2014 11th International Conference on Security and Cryptography (SECRYPT)* (2014), pp. 1–9.

[226] Jonas Wagner, Simon Müller, Christian Näther, Jan-Philipp Steghöfer, and Andreas Both. “Towards Effective Complementary Security Analysis Using Large Language Models”. In: *2025 IEEE International Conference on Intelligence and Security Informatics (ISI)* (2025), pp. 21–28. URL: https://api.semanticscholar.org/CorpusID:279465216.

[227] David Waltermire, Paul R. Cichonski, and Karen Scarfone. “Common Platform Enumeration: Applicability Language Specification Version 2.3”. In: 2011.

[228] Junjie Wang, Song Wang, and Qing Wang. “Is there a "golden" feature set for static warning identification?: an experimental evaluation”. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2018). URL: https://api.semanticscholar.org/CorpusID:52942461.

[229] Emil Wåreus and Martin Hell. “Automated CPE Labeling of CVE Summaries with Machine Learning”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment* 12223 (2020), pp. 3–22.

[230] Sam Weber, Paul A. Karger, and Amit M. Paradkar. “A software flaw taxonomy”. In: *ACM SIGSOFT Software Engineering Notes* 30 (2005), pp. 1–7.

[231] Ying Wei, Lili Bo, Xiaobing Sun, Bin Li, Zhang Tao, and Chuanqi Tao. “Automated event extraction of CVE descriptions”. In: *Inf. Softw. Technol.* 158 (2023), p. 107178.

[232] Ying Wei, Xiaobing Sun, Lili Bo, Sicong Cao, Xin Xia, and Bin Li. “A comprehensive study on security bug characteristics”. In: *Journal of Software: Evolution and Process* 33 (2021). URL: https://api.semanticscholar.org/CorpusID:238254597.

[233] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. “Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We?” In: *ACM Transactions on Knowledge Discovery from Data* 18 (2024), pp. 1–34.

[234] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Gülsüm Acar, and Sascha Fahl. “"Always Contribute Back": A Qualitative Study on Security Challenges of the Open Source Supply Chain”. In: *2023 IEEE Symposium on Security and Privacy (SP)* (2023), pp. 1545–1560. URL: https://api.semanticscholar.org/CorpusID:259373507.

[235] David Wheeler. *Flawfinder Documentation*. https://dwheeler.com/flawfinder/flawfinder.pdf. [Online; accessed 23-December-2022]. 2017.

[236] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. “A Survey on Software Fault Localization”. In: *IEEE Transactions on Software Engineering* 42 (2016), pp. 707–740.

[237] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. “Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities”. In: *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2024).

[238] Yan Wu, Robin A. Gandhi, and Harvey P. Siy. "Using semantic templates to study vulnerabilities recorded in large software repositories". In: *SESS '10*. 2010.

[239] Dapeng Yan, Yuqing Niu, Kui Liu, Zhe Liu, Zhiming Liu, and Tégawendé F. Bissyandé. "Estimating the Attack Surface from Residual Vulnerabilities in Open Source Software Supply Chain". In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)* (2021), pp. 493–502.

[240] Aidan Z. H. Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. "Security Vulnerability Detection with Multitask Self-Instructed Fine-Tuning of Large Language Models". In: *ArXiv* abs/2406.05892 (2024). URL: https://api.semanticscholar.org/CorpusID:270371828.

[241] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. "KNighter: Transforming Static Analysis with LLM-Synthesized Checkers". In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (2025). URL: https://api.semanticscholar.org/CorpusID:276938308.

[242] Guanqun Yang, Shay Dineen, Zhipeng Lin, and Xueqing Liu. "Few-Sample Named Entity Recognition for Security Vulnerability Reports by Fine-Tuning Pre-Trained Language Models". In: *ArXiv* abs/2108.06590 (2021).

[243] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. "Learning to recognize actionable static code warnings (is intrinsically easy)". In: *Empirical Software Engineering* 26 (2020). URL: https://api.semanticscholar.org/CorpusID:225093746.

[244] Yixin Yang, Ming Wen, Xiang Gao, Yuting Zhang, and Hailong Sun. "Reducing False Positives of Static Bug Detectors Through Code Representation Learning". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2024), pp. 681–692. URL: https://api.semanticscholar.org/CorpusID:271246675.

[245] He Ye, Zimin Chen, and Claire Le Goues. "PreciseBugCollector: Extensible, Executable and Precise Bug-Fix Collection: Solution for Challenge 8: Automating Precise Data Collection for Code Snippets with Bugs, Fixes, Locations, and Types". In: *2023 38th International Conference on Automated Software Engineering (ASE)* (2023), pp. 1899–1910. URL: https://api.semanticscholar.org/CorpusID:261696581.

[246] Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo, and Tim Menzies. "How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs". In: *IEEE Transactions on Software Engineering* 49 (2022), pp. 2856–2872. URL: https://api.semanticscholar.org/CorpusID:255096745.

[247] Sai Yerramreddy, Austin Mordahl, Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. "An empirical assessment of machine learning approaches for triaging reports of static analysis tools". In: *Empirical Software Engineering* 28 (2023), pp. 1–44. URL: https://api.semanticscholar.org/CorpusID:255693219.

[248] Sofonias Yitagesu, Zhenchang Xing, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Linyi Han. "Extraction of Phrase-based Concepts in Vulnerability Descriptions through Unsupervised Labeling". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[249] Ulas Yuksel and Hasan Sözer. "Automated Classification of Static Code Analysis Alerts: A Case Study". In: *2013 IEEE International Conference on Software Maintenance* (2013), pp. 532–535. URL: https://api.semanticscholar.org/CorpusID:15609238.

[250] Michal Zalewski. *American Fuzzy Lop (AFL)*. https://lcamtuf.coredump.cx/afl/. Accessed: 2026-02-17.

[251] Francisco Zanartu, Christoph Treude, Bruno Cartaxo, Hudson Silva Borges, Pedro Moura, Markus Wagner, and Gustavo Henrique Lima Pinto. "Automatically Categorising GitHub Repositories by Application Domain". In: *ArXiv* abs/2208.00269 (2022). URL: https://api.semanticscholar.org/CorpusID:251224407.

[252] Andreas Zeller. "Isolating cause-effect chains from computer programs". In: *SIGSOFT '02/FSE-10*. 2002.

[253] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. "On the impact of security vulnerabilities in the npm and RubyGems dependency networks". In: *Empirical Software Engineering* 27 (2021). URL: https://api.semanticscholar.org/CorpusID:235421650.

[254] Lyu Bing Zhang, Jiahui Wu, Chengwei Liu, Kaixuan Li, Xiaoyu Sun, Lida Zhao, Chong Wang, and Yang Liu. "Fixing Outside the Box: Uncovering Tactics for Open-Source Security Issue Management". In: *Proceedings of the ACM on Software Engineering* 2 (2025), pp. 2273–2295. URL: https://api.semanticscholar.org/CorpusID:277451787.

[255] Su Zhang, Doina Caragea, and Xinming Ou. "An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities". In: *International Conference on Database and Expert Systems Applications*. 2011.

[256] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jianyun Nie, and Ji-rong Wen. "A Survey of Large Language Models". In: *ArXiv* abs/2303.18223 (2023).

[257] Yangyang Zhao, Hareton K. N. Leung, Yibiao Yang, Yuming Zhou, and Baowen Xu. "Towards an understanding of change types in bug fixing code". In: *Inf. Softw. Technol.* 86 (2017), pp. 37–53. URL: https://api.semanticscholar.org/CorpusID:205437589.

[258] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. "Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection". In: *ArXiv* abs/2407.16235 (2024). URL: https://api.semanticscholar.org/CorpusID:271334648.

[259] Xin Zhou, Ting Zhang, and David Lo. "Large Language Model for Vulnerability Detection: Emerging Results and Future Directions". In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. Association for Computing Machinery, 2024.

[260] Yaqin Zhou, Shangqing Liu, J. Siow, Xiaoning Du, and Yang Liu. "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks". In: *ArXiv* abs/1909.03496 (2019).

[261] Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. "CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities". In: *ArXiv* abs/2503.17332 (2025). URL: https://api.semanticscholar.org/CorpusID:277244869.

[262] Noah Ziems and Shaoen Wu. "Security Vulnerability Detection Using Deep Learning Natural Language Processing". In: *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2021), pp. 1–6. URL: https://api.semanticscholar.org/CorpusID:233864893.

[263] Jason Zwolak. *Program Bug Examples*. Accessed 21-October-2020. Oct. 2000. URL: http://courses.cs.vt.edu/~cs1206/Fall00/bugs_CAS.html.

Appendix A

Implementation Details (Chapter 2)

This appendix provides the detailed implementation procedures supporting Chapter 2. It is organized into two main parts. Section A.1 documents the dataset construction pipeline, including metadata filtering, consistency checks, repository resolution, description extraction, and patch validation steps used to derive the curated corpus from raw NVD records. Section A.2 specifies the taxonomies and mapping rules used for profiling, covering root-cause labeling, error and failure normalization, and the contextual classification of vulnerable software. Together, these sections formalize the concrete steps to transform raw vulnerability records into structured, comparable representations suitable for analysis.

A.1 Dataset Construction: Filtering and Curation

This section summarizes the sequential filtering and curation pipeline used to derive a high-integrity vulnerability dataset from raw NVD records for bug–error–failure analysis. The process begins by normalizing entry status and tag metadata to remove invalid or ambiguous records (Section A.1.1), followed by CPE dictionary processing to enforce consistent, application-level product identification (Section A.1.2) and the application of CWE eligibility constraints to retain only well-defined and allowed weakness classes (Section A.1.3). CVSSv3 metrics are then examined for completeness and internal consistency, and additional scoring constraints are imposed to ensure stable and comparable impact characteristics (Sections A.1.4 and A.1.5). Surviving CVEs are mapped to canonical open-source repositories through CPE- and reference-based validation (Section A.1.6), after which vulnerability descriptions are screened for extractable manifestation entities and corresponding patches are resolved and validated. Only entries that yield a concrete, well-scoped code change suitable for root-cause reconstruction are retained in the final dataset (Sections A.1.7 and A.1.8).

A.1.1 NVD Status and Tag Filters

This appendix documents the filtering rules applied to CVE records based on NVD analysis status and assigned tags. Although both invalid and incomplete records are excluded from the curated

corpus, we distinguish between these categories explicitly, as they are reported separately in the filtering breakdown shown in Fig. 2.4.

Accepted and excluded NVD statuses. Each CVE in the NVD feed is associated with a status indicating the state of NVD analysis. We retain only CVEs whose status reflects a completed analysis, namely those marked as *Analyzed* or *Modified*. Excluded statuses are divided into two categories. CVEs with status *REJECTED* are treated as *invalid*, as the corresponding identifiers have been explicitly withdrawn or determined to be erroneous. CVEs in provisional or unfinished states—*RECEIVED*, *AWAITING_ANALYSIS*, *UNDERGOING_ANALYSIS*, and *DEFERRED*—are treated as *incomplete*, since their metadata may be partial, unstable, or subject to substantial future revision. While both categories are excluded from downstream analysis, this distinction is preserved for reporting and visualization purposes.

Excluded tags. Independently of status, we filter CVEs based on the presence of specific NVD tags that indicate semantic ambiguity or limited interpretability. CVEs tagged as *disputed* are classified as *invalid*, reflecting unresolved disagreement regarding the vulnerability's validity or scope. CVEs tagged as *unsupported-when-assigned*, indicating that the affected product was already end-of-life at the time of CVE assignment, are likewise classified as *invalid*, as such cases typically lack meaningful remediation or software-context signals. The presence of any excluded tag is sufficient to disqualify a CVE.

A.1.2 CPE Dictionary Processing and Consistency Checks

This appendix describes how CPE metadata is parsed, normalized, and validated to ensure that CVE platform annotations can be reliably mapped to coherent software products. CPE loading, grouping, and validation are applied during ingestion, prior to any CVE–CPE association checks.

CPE dictionary snapshot and parsing. We use the official NVD CPE Dictionary in JSON format (CPE specification version 2.3), corresponding to the snapshot dated **29–12–2025**. JSON feeds are used exclusively, as the XML representation has been deprecated by NVD. All entries are parsed using a custom Python library developed for this work, `cpelib`, which provides structured access to CPE fields and preserves dictionary-provided metadata such as deprecation flags and references.

Vendor–product grouping and consistency checks. All parsed CPE entries are grouped by their *(vendor, product)* tuple, collapsing version- and configuration-specific CPEs into a single product-level entity. From 1,491,866 CPE entries, this yields 143,002 unique vendor–product pairs. Validation is performed at this grouped level, and certain pairs are excluded based on two disjoint conditions:

- **Deprecated products.** A vendor–product pair is labeled as *deprecated* if all associated CPE entries are marked as deprecated in the dictionary. Such products are excluded, as they no longer represent actively referenceable software artifacts.
- **Inconsistent products.** A vendor–product pair is labeled as *inconsistent* if its associated CPE entries span multiple incompatible CPE part types (i.e., if the set of part values has cardinality greater than one). In particular, mixtures of *application*, *operating system*, and *hardware* within a single vendor–product group are treated as semantic inconsistencies and excluded to prevent ambiguous software-type attribution.

Filtering and labeling outcome. Across the dictionary snapshot used in this study, 3,644 vendor–product pairs are classified as deprecated or inconsistent and removed from the eligible CPE set. Each remaining vendor–product pair is assigned a single normalized product label—*application*, *operating system*, or *hardware*—derived from its CPE part type. This guarantees a unique and internally consistent product classification for all CPEs propagated to downstream analyses.

A.1.3 CWE Eligibility Rules

This appendix specifies the rules used to determine whether a Common Weakness Enumeration (CWE) entry is eligible for CVE root-cause profiling. Although many CVEs are associated with one or more CWE identifiers, not all CWE entries are suitable for consistent and interpretable root-cause analysis. Extremely abstract entries, deprecated definitions, and CWEs discouraged by MITRE guidance can obscure the initiating defect rather than clarify it. CWE eligibility is evaluated prior to downstream CVE profiling.

CWE catalog and eligibility criteria. We use the official CWE List version 4.18, obtained in XML format from MITRE and parsed into structured objects using the `pydantic-cwe` Python package.[1] Eligibility is determined directly from catalog attributes, and a CWE is retained if and only if it satisfies all of the following conditions:

- **Active definition.** CWEs marked as *Deprecated* are excluded to avoid references to superseded or withdrawn definitions.
- **Allowed vulnerability mapping.** CWEs whose mapping notes specify a usage category other than *Allowed* or *Allowed-with-Review* are excluded, ensuring that retained CWEs are recommended by MITRE for vulnerability mapping.
- **Permitted abstraction level.** Only CWEs with abstraction level *Variant*, *Base*, or *Class* are retained. Compound, chain, and other composite abstractions are excluded, as they conflate root causes with downstream manifestations.

[1] `pydantic-cwe`: https://pypi.org/project/pydantic-cwe/

Handling of multiple and placeholder CWEs. A CVE may reference multiple CWE identifiers, including placeholder values (e.g., *CWE-Other* or *CWE-noinfo*). CVEs with no associated CWE entries are treated as *incomplete* and excluded. For CVEs with one or more CWE identifiers, the CVE is retained if at least one referenced CWE satisfies the eligibility criteria above.

Categorization and filtering outcome. Each retained CWE is mapped to MITRE's *Comprehensive Categorization for Software Assurance Trends* view to support topic-level aggregation and comparison. Applying the eligibility rules yields 837 eligible CWEs out of 969 total entries in the catalog. Category-level comparisons before and after filtering show that the criteria primarily remove overly generic or discouraged entries while preserving the overall topical structure. Several categories—Comparison, Component Interaction, Incorrect Calculation, Protection Mechanism Failure, and Resource Control—experience no removals, while the largest reductions occur in Poor Coding Practices (141 → 90), Resource Life Cycle Management (106 → 92), and Access Control (155 → 148), as illustrated in Fig. A.1.

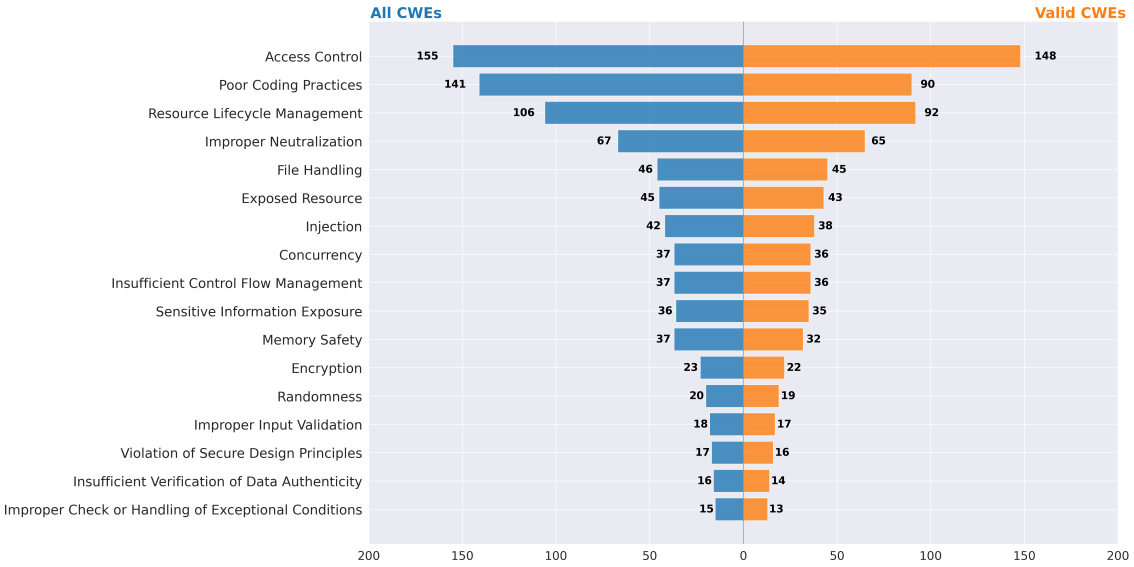


Figure A.1: CWE Counts Before and After Filtering by Software Assurance Trends Categories.

A.1.4 CVSSv3 Inconsistency Measure

This appendix defines the metric used to detect and exclude CVEs with inconsistent CVSSv3 assessments. While CVSSv3 scores are intended to provide a stable characterization of exploitability and impact, some CVEs contain multiple assessments from different sources whose qualitative metrics diverge. To ensure a precision-oriented corpus, we quantify such divergence and exclude CVEs whose assessments are not fully aligned. CVSSv3 inconsistency is evaluated after status, CPE, and CWE filtering, and all computations are performed using structured CVSSv3 objects provided by the `nvdutils` Python package.

Pairwise inconsistency computation. Given two CVSSv3 assessments A and B for the same CVE, we compute a pairwise inconsistency score by comparing their base and impact metrics. Each qualitative CVSSv3 metric is mapped to a numerical value reflecting relative severity or likelihood, using the standard metric values defined in the CVSSv3 specification [2]. These values are treated as fixed and comparable across assessments. Scope is handled separately. If the Scope values differ between the two assessments, a penalty of 1 is added; otherwise, no penalty is applied. For the *Privileges Required* metric, different numeric mappings are used depending on whether Scope is *Changed* or *Unchanged*, in accordance with the CVSSv3 specification. Let M denote the set of aligned base and impact metrics excluding Scope, and let v_m^A and v_m^B be the numeric values of metric $m \in M$ under assessments A and B. For each metric, we compute the normalized difference d_m as shown in Eq. (A.1). The pairwise inconsistency score $I(A,B)$ is then defined in Eq. (A.2), yielding a value in the interval $[0,1]$.

$$d_m = \frac{|v_m^A - v_m^B|}{\max(v_m) - \min(v_m)} \quad \text{(A.1)} \qquad I(A,B) = \frac{\mathbb{1}[\text{Scope}_A \neq \text{Scope}_B] + \sum_{m \in M} d_m}{|M| + 1} \quad \text{(A.2)}$$

Aggregation, threshold rule, and exclusion semantics. If a CVE contains more than two CVSSv3 assessments, all pairwise inconsistency scores are computed and averaged. If all assessments share an identical CVSS vector string, the inconsistency score is defined to be 0. CVEs with no CVSSv3 assessments are treated as *incomplete*. We apply a strict threshold: CVEs with a CVSSv3 inconsistency score strictly greater than 0 are excluded and classified as *invalid*. This corresponds to requiring exact agreement across all qualitative CVSSv3 base and impact metrics, including Scope, among all available assessments. This zero-tolerance rule reflects a deliberate precision-over-recall design choice.

Worked example. Consider a CVE with two CVSSv3 assessments that differ only in Attack Vector: one specifies *NETWORK* (0.85) and the other *LOCAL* (0.55), while all other metrics and Scope agree. The normalized difference for Attack Vector is

$$\frac{0.85 - 0.55}{0.85 - 0.20} \approx 0.46.$$

With no Scope penalty and seven total metrics, the resulting inconsistency score is non-zero, and the CVE is therefore excluded under the threshold rule.

A.1.5 CVSSv3 Low-Contingency and Failure-Boundary Filters

This appendix specifies the CVSSv3 constraints used to identify vulnerabilities whose activation depends on few external contingencies and whose failure effects remain well-bounded and comparable. Together, these filters operationalize a restricted execution and failure model by selecting

[2] https://www.first.org/cvss/v3-0/specification-document#8-4-Metrics-Levels

CVEs that are remotely reachable, input-driven, independent of user actions and privileges, confined to a single authorization scope, and characterized by a consistent impact regime.

CVSSv3 constraints and operational rationale. For each CVE with a valid and internally consistent CVSSv3 assessment (Appendix A.1.4), we apply the constraints summarized in Table A.1; failure to satisfy any single condition disqualifies the CVE.

Table A.1: CVSSv3 constraints used to select vulnerabilities.

Metric	Required Value	Operational Rationale
Attack Vector (AV)	NETWORK	Ensures remote reachability without reliance on local access or physical proximity.
User Interaction (UI)	NONE	Removes dependencies on user behavior, enabling purely input-driven triggering.
Privileges Required (PR)	NONE	Avoids assumptions about privileged execution contexts.
Attack Complexity (AC)	LOW	Excludes vulnerabilities requiring rare timing, races, or complex environmental setup.
Scope (S)	UNCHANGED	Constrains consequences to the same authorization scope as the vulnerable component, preserving a self-contained failure boundary.
Base Severity	CRITICAL	Normalizes the corpus to a consistent impact regime, avoiding qualitative mixing of severity levels.

A.1.6 OSS Repository Resolution and Application-Level Traceability

This appendix documents how curated CVEs are restricted to open-source, application-level components for which bug-to-failure chains can be reconstructed from public code artifacts. The procedure builds on the CPE normalization and product dictionary construction described in Appendix A.1.2, with the additional constraint that only CPE entries whose *part* field is marked as application (`part = "a"`) are retained. Records referencing operating systems (`"o"`) or hardware (`"h"`) are excluded, as these broaden the failure boundary beyond a single component.

Repository linkage sources and precedence. For each retained (vendor, product) pair, we attempt to resolve a corresponding GitHub repository using two complementary linkage signals. The primary source is the SCANOSS `purl2cpe` alignment dataset (version `autodump_20251230` [3]), which maps CPE identifiers to ecosystem Package URLs (PURLs). Only PURLs of type `github` are retained, yielding candidate GitHub repositories associated with each product. From 2,356,712 CPE–PURL pairs, this process produces 20,016 unique vendor–product–repository mappings for GitHub-hosted projects. Direct repository references embedded in CPE metadata are used as a

[3] https://github.com/scanoss/purl2cpe/commit/095fd458b27c7b9df7053db745f8e2ca6c74db4a

secondary signal. Reference URLs tagged as *Product*, *Project*, or *Version* with `github.com` are parsed to recover canonical *owner/repository* identifiers. These direct mappings are added only when no repository has already been established for the corresponding product via the `purl2cpe` crosswalk, ensuring a single canonical repository per product while avoiding source conflicts.

Repository resolvability. All candidate repositories are validated using the `github-lib` [4] Python package, which interfaces with the GitHub API to confirm repository existence, accessibility, and canonical ownership. Validation resolves redirects, normalizes owner and repository identifiers, and excludes repositories that are unavailable, deleted, or inaccessible at query time. Forks are accepted only when they correspond to the canonical upstream project as determined by GitHub metadata. A product is classified as *resolvable* if a stable GitHub owner and repository identifier can be confirmed through this process. Products whose repositories cannot be located or validated are labeled *unresolvable* and excluded from downstream analysis. Applying these checks yields 14,876 resolvable open-source application products out of 15,329 GitHub-linked application entries, with 454 repositories excluded due to resolution or availability failures.

CVE-level OSS inclusion. At the CVE level, repository resolution is evaluated across all affected products listed in the vulnerability configuration. A CVE is retained for OSS-based analysis if it affects at least one valid, non-deprecated product that maps to a resolvable open-source repository. CVEs whose affected products are exclusively invalid, unresolved, or non-application components are excluded. Additional configuration-level properties (e.g., multi-product and multi-part cases) are recorded for later analysis and discussed in the subsequent appendix subsection.

A.1.7 Quality Screening of Descriptions and Extraction of Vulnerability Entities

This appendix details the screening and parsing steps applied to CVE descriptions to determine whether they provide sufficient and reliable manifestation cues for reconstructing bug–error–failure chains. The procedure combines lightweight linguistic quality checks with deterministic, pattern-based extraction rules, deliberately favoring precision and interpretability over coverage.

Description eligibility and quality screening. Each CVE is evaluated using its English-language description as reported in the NVD. Before attempting extraction, descriptions are subjected to a sequence of exclusion checks designed to remove cases that are structurally ambiguous, linguistically unreliable, or underspecified. First, descriptions that explicitly refer to multiple vulnerabilities (e.g., phrases such as “multiple buffer overflows” or “several vulnerabilities”) are excluded, as they do not support a one-to-one mapping between a defect and its manifestation. This is detected using a targeted regular expression that matches pluralized vulnerability constructions while guarding against overly broad matches. Second, only single-sentence descriptions are retained. Sentence segmentation is performed using NLTK’s standard sentence tokenizer, and descriptions

[4] `github-lib` - https://pypi.org/project/github-lib/

containing more than one sentence are excluded to avoid cases where multiple failure modes, conditions, or impacts are conflated within the same entry. Third, the remaining descriptions are filtered using a linguistic acceptability classifier [5] based on a RoBERTa model fine-tuned on the CoLA benchmark. This classifier estimates the grammatical well-formedness of each sentence; descriptions with an acceptability score below a fixed threshold (0.5) are discarded. This step functions as a pragmatic sanity check to remove malformed, truncated, or syntactically incoherent descriptions that would undermine rule-based parsing. Descriptions passing all three checks are considered *compliant* and eligible for downstream extraction.

Pattern matching and extraction strategy. Eligible descriptions are processed using a deterministic pattern-matching pipeline implemented in the `cve-driller` [6] library. The extraction logic is based on a curated set of regular expressions designed to capture recurring linguistic templates observed in CVE descriptions. These patterns operationalize the vulnerability concept taxonomy introduced by Evans et al. [161], with minor adaptations to align with CPE terminology (e.g., renaming *vuln_type* to *weakness*). Extraction is organized hierarchically (depicted in Figure A.2). Each description is first matched against a small set of generic entry patterns; successful matches trigger increasingly specific templates ordered from information-rich to minimal forms. This strategy maximizes the amount of structured information recovered when available, while still allowing partial extraction in cases where descriptions are concise but informative. A description is considered *matched* if at least one extraction pattern applies.

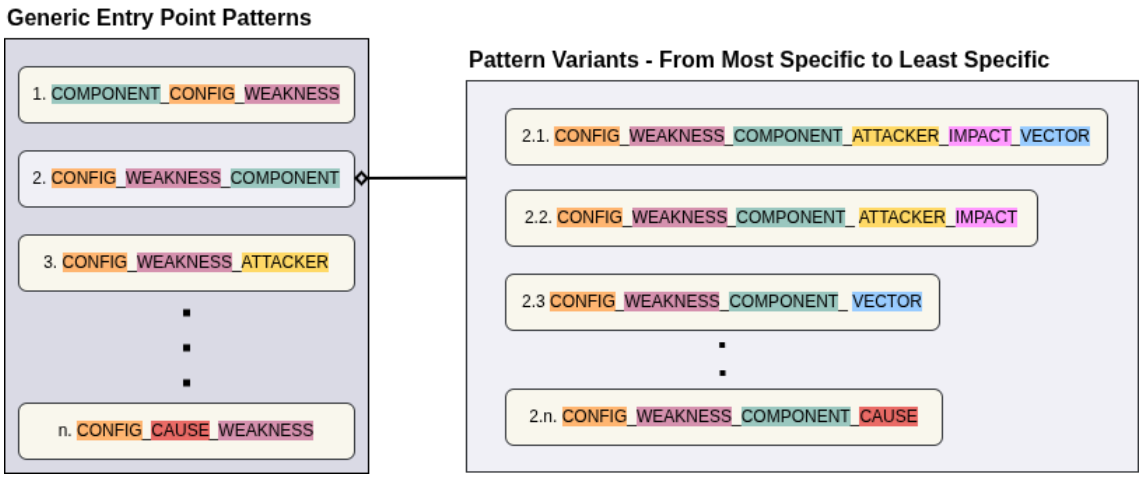


Figure A.2: Hierarchical Regular Expression Extraction Strategy

Matches and extracted entities. When a match occurs, the extractor returns a structured record containing the subset of vulnerability concepts supported by the matched pattern. These may include *weakness*, *impact*, *component*, *attacker*, *vector*, and *configuration*. All extraction rules are deterministic and rely on explicit capture groups within the regular expressions; no probabilistic

[5] `roberta-base-CoLA` - https://www.promptlayer.com/models/roberta-base-cola
[6] `cve-driller` - https://pypi.org/project/cve-driller/

inference or learned post-processing is applied. Cases that match patterns but do not yield both weakness and impact entities are excluded, as they do not provide sufficient information to anchor both the error and failure stages of the chain. Applying this screening and extraction pipeline yields 924 matched descriptions from the compliant subset, of which 451 contain both weakness and impact entities and are retained for downstream analysis.

A.1.8 Patch Resolution and Validation

This stage operates on the subset of CVEs that pass description screening and entity extraction (Appendix A.1.7) and aims to identify a concrete, accessible patch that can serve as evidence for the root-cause bug. Only CVEs for which a valid patch can be unambiguously resolved and inspected are retained for downstream bug–error–failure reconstruction.

Candidate patch identification from references. Patch discovery is driven by the `references` field of each CVE record. We inspect all reference URLs and retain only those that resolve to GitHub resources, including direct commit and issue URLs that can be deterministically mapped to a fixing commit. Non-GitHub links (e.g., advisories, mailing lists, release notes) are ignored. For each CVE, candidate references are evaluated against the set of previously resolved OSS products (Appendix A.1.6). A reference is considered eligible only if the resolved GitHub repository matches the affected product's canonical repository, as determined by stable `owner_id` and `repo_id` identifiers. This prevents cross-project contamination when a CVE references multiple repositories or third-party forks. Issue URLs are followed to retrieve the associated fixing commit when available; direct commit URLs are resolved by commit hash. References that do not yield a commit object are discarded.

Patch availability, exclusion, and filtering. A CVE is considered to have a *resolvable patch* if at least one reference yields a commit that: (i) exists in the validated repository, (ii) is accessible via the GitHub API, and (iii) can be retrieved without ambiguity. CVEs are excluded at this stage if all candidate references fail due to one or more of the following conditions: deleted or renamed repositories, private or inaccessible forks, missing commits, repository–product mismatches, or unresolved issue-to-commit mappings. To improve robustness and reduce redundant API calls, resolved repositories, commits, and failed lookups are cached; repositories previously confirmed as unavailable are not re-queried. For each resolved commit, we retrieve the full diff and apply additional validation to ensure that the patch provides meaningful bug evidence. Only diffs affecting source files in a predefined set of target programming languages are retained; non-code files and test-only changes are excluded. Commits that modify an excessive number of lines (above a fixed threshold) are discarded to avoid large refactorings or mixed-purpose changes that obscure the bug. If no qualifying diff files remain after this filtering, the CVE is excluded from analysis.

Patch summarization and root-cause selection. Validated diffs are summarized at the file level using a dedicated diff-to-text language model [7]. Each file-level summary is parsed into a structured form consisting of a high-level fix type (e.g., *bugfix*, *security*) and a natural-language description of the changes. When multiple candidate files are present, the most probable root-cause location is selected by combining (i) semantic similarity between the original commit message and the generated summaries, and (ii) a fixed prior over fix types that favors security-relevant and corrective changes. If no valid summary can be produced, the commit message itself is retained as a fallback root-cause description. CVEs for which no patch passes all resolution, validation, and summarization steps are classified as having no resolvable patch and are excluded from the corpus. In practice, these summaries are not always semantically faithful to the underlying change and are therefore used as auxiliary signals rather than as direct representations of the bug.

A.2 Profiling Labels: Taxonomies and Mappings

This section details the taxonomies and mapping rules used to implement the profiling in Section 2.4.5. It specifies the patch-grounded scheme for labeling the bug cause and operation (Section A.2.1), the normalization of CWE and impact statements into error and failure categories (Sections A.2.2 and A.2.3), and the contextual taxonomies used to classify software type and functional area (Sections A.2.4 and A.2.5). Together, these definitions operationalize the root cause, manifestation, and software context dimensions used for cross-case comparison.

A.2.1 Root cause taxonomy (cause × operation) and patch evidence

This subsection defines the labeling scheme and patch-handling rules used to assign a *root cause* to each vulnerability. The scheme adopts categories primarily from the NIST's BF, supplemented by CWE terminology, and characterizes each bug as a compact *Cause* × *Operation* pair (e.g., Missing Validate, Incorrect Evaluate). The goal is not to introduce a new taxonomy, but to normalize diverse fixes into a consistent and comparable representation grounded in established frameworks. Because labels are derived from code changes, the subsection also specifies how patches are selected and reduced to the specific code change used for labeling.

Bug cause. The bug cause is classified with two mutually exclusive qualifiers: *Missing* and *Incorrect*. This distinction is derived from MITRE's CWE glossary, which separates behaviors that are not attempted at all from those that are attempted but implemented incorrectly. In CWE terminology, *Improper* is a broad that can subsume both Missing and Incorrect, while *Insufficient* captures a graded form of incorrectness. For the purposes of bug normalization, these notions are reduced to a binary distinction:

- **Missing** denotes cases where the developer does not attempt the relevant behavior at all (e.g., no bounds check, no authorization check, no validation logic).

[7] https://huggingface.co/hks350d/git-diff-to-commit-gemma-3-270m

- **Incorrect** denotes cases where the behavior is present but flawed (e.g., incomplete validation, incorrect condition, wrong calculation, unsafe API usage).

This reduction avoids ambiguity introduced by context-dependent notions of sufficiency while preserving the core distinction needed for cross-case comparison.

Operation categories. To characterize what program action is missing or incorrect, each case is assigned a single *operation* label identifying the minimal code-level action whose omission or misimplementation enables the vulnerability. The taxonomy is grounded primarily in NIST's BF, which defines security bugs in terms of execution-phase operations. In this work, these operations are interpreted at the code level by identifying the concrete program action (e.g., checking, calculating, allocating) that realizes the corresponding behavior in the implementation. When the operation is not directly observable in the vulnerable code—e.g., when it is implicit in assumptions or only becomes explicit in the fix; CWE terminology is used to establish alignment by selecting the closest weakness class and mapping it to the corresponding BF operation. A CWE is recorded for each case, both for consistency and to support this alignment when needed. The operation categories used in this thesis include:

- VALIDATE: checking data syntax or structure (e.g., format, presence of required elements).
- SANITIZE: modifying data to make it safe or well-formed (e.g., escaping, filtering).
- VERIFY: checking data semantics or meaning (e.g., value ranges, logical consistency).
- DECLARE: specifying names, types, or constraints of variables, functions, or data structures.
- CALCULATE: computing numeric, pointer, or string results.
- EVALUATE: evaluating boolean conditions or comparisons.
- INITIALIZE: the first write into an object, after it is allocated.
- READ: gets content from an object.
- WRITE: puts content into an object.
- CLEAR: the very last write into an object, before it is deallocated.
- DEREFERENCE: overreaches INITIALIZE, READ, WRITE, and CLEAR; the focus is on object access regardless of whether it is for reading or writing.

For higher-level synthesis, individual operation labels are further grouped according to the models defined in the NIST's BF [19]. Specifically, VALIDATE, SANITIZE, and VERIFY are aggregated under the *Data Check* category (corresponding to BF's Input/Output model), which addresses temporal input/output safety by prescribing the correct sequencing of data checking operations (e.g., ensuring that input is validated and sanitized prior to use). Likewise, DECLARE,

EVALUATE, and CALCULATE are grouped under the *Data Type* category (corresponding to BF's Data Type model), which addresses temporal type safety by prescribing correct declaration and computation semantics (e.g., preventing wrap-around through proper type specification). Finally, INITIALIZE, READ, WRITE, CLEAR, and DEREFERENCE are grouped under the *Memory* category (corresponding to BF's Memory model), which captures operations that act on the content or validity of objects in memory across their allocation lifetime. These higher-level categories are used for aggregation only and do not replace the Cause $\times$ Operation label. Readers are referred to the official BF documentation for full model definitions and formalization details [18].

Decision boundaries and disambiguation. When multiple operations appear in a fix, the operation that most directly explains the manifestation described in the CVE is selected. For example, adding a conditional guard that enforces an input constraint is labeled as VALIDATE rather than DEREFERENCE, unless the vulnerability arises specifically from accessing an invalid object. Similarly, patches that combine declaration changes with downstream checks are labeled based on the operation whose absence or mistake made the exploit possible. When the bug-relevant action is accessing an object through a pointer or reference—regardless of whether that access is for reading, writing, passing as an argument, or recursing into—and the defect is that the object may no longer be valid at that point, DEREFERENCE is used. READ or WRITE are preferred only when the access is valid, and the bug lies in what value is read or written, not in whether the object is still live. The operation is attached to the source construct that *enables* invalid state, not to the downstream crash or memory-safety failure site: the crash site identifies the manifestation; the root-cause construct is the earlier statement or design decision that allowed the object to become invalid before the access occurs.

Selecting patch evidence ("the fix"). Root cause labeling is anchored in patch artifacts. References in NVD records may point to issues, pull requests, release notes, or multiple commits. For each CVE, *the fix* is identified as the commit (or the most relevant commit within a pull request) that best explains the vulnerability described. Concretely, this is the change that introduces, corrects, or enforces the behavior implicated by the description—such as adding a missing check on the affected input or correcting an erroneous condition. Surrounding refactors are ignored unless they are directly tied to the bug. If validation or enforcement logic is moved into a helper function or relocated across files as part of the fix, the most direct path to the bug is followed: typically the vulnerable file when identifiable, otherwise the nearest call site whose change enforces the missing or corrected operation. Multi-commit fixes are reduced to the commit that introduces the decisive change.

Labeling protocol and consistency notes. Root cause labeling was performed manually using a fixed protocol: (i) read the CVE description to identify the manifestation, (ii) inspect the selected patch diff to identify the security change that addresses the bug, and (iii) assign a single

cause label and operation label using the decision boundaries above. Ambiguous cases were resolved through written disambiguation notes and revisited during consistency checks. Periodic spot reviews were conducted to ensure stable interpretation of operation boundaries and consistent treatment of refactors, helper extraction, and partial fixes.

A.2.2 Error taxonomy and CWE mapping

This subsection defines the error taxonomy used to label the *manifestation* side of reconstructed vulnerability chains and documents how CWE identifiers are mapped into normalized error categories. The objective is to abstract heterogeneous CWE assignments into a small set of error labels that characterize the *runtime semantic deviation* induced by a bug, enabling consistent cross-case comparison across heterogeneous vulnerabilities.

Conceptual basis. The taxonomy is grounded in classical dependability and security theory, in which system behavior is analyzed through the fault–error–failure chain [10]. In this model, a *fault* (or defect) is a static cause, an *error* is a deviation of system state or behavior at runtime, and a *failure* is an externally observable deviation of delivered service. This taxonomy explicitly targets the *error* level. Its design draws on three complementary perspectives:

- Dependability theory [10], which characterizes errors as invalid or inconsistent runtime states that may lead to failure.
- Bazaz and Arthur's process/object model [12], which explains vulnerabilities in terms of software processes interacting with computational resources under resource-imposed constraints and process-side assumptions.
- MITRE's vulnerability theory and CWE guidance [162], which distinguish root causes, affected resources, and observable manifestations.

This taxonomy inherits Bazaz and Arthur's process/resource abstraction but changes the classification target. Whereas Bazaz and Arthur classify *vulnerabilities* as violable constraints and assumptions associated with resources, this taxonomy classifies the *runtime error manifestations* that arise when such vulnerabilities are exercised. Its purpose is therefore not to explain why a process is vulnerable, but to normalize how its semantics deviate at runtime.

Runtime abstraction. Following Bazaz and Arthur's process/object model of computation [12], a software application at runtime is viewed as an executing process that accepts input, performs operations, and produces output while using computational resources through the operating system interface. In their model, the main resource classes are *main memory*, *input/output (I/O)*, and, more peripherally for our purposes, *cryptographic resources*. For the purposes of error analysis, we restate this process/resource view in a form that makes the manifestation side explicit:

- An *internal state*, corresponding primarily to Bazaz and Arthur's *main memory* resource, comprising the process state held in memory, including stored values, control state, and execution invariants.
- A set of *interfaces*, corresponding to the process's use of *I/O* to receive input and produce output, through which data enters and leaves the component.
- A set of *external resources*, also derived from their *I/O* view of process interaction with the environment, comprising resources the process accesses via those interfaces, such as files, databases, devices, interpreters, or peer services.

This restatement does not change Bazaz and Arthur's underlying abstraction; it makes explicit, for error analysis purposes, the distinction between the process's internal operating state and the two main ways it interacts with its environment: exchanging information across interfaces and acting upon external resources. Cryptographic resources remain part of the original model, but are not a primary focus of the present taxonomy.

Minimal semantic core of error deviations. From this process/resource abstraction, reinterpreted at the error level, we derive a minimal and non-overlapping set of error categories:

- **State Corruption**: violation of the integrity of the component's internal state, whereby stored values, control state, or invariants become incorrect, inconsistent, or unsafe.
- **Authority Violation**: breach of authority boundaries governing the component's interaction with external resources, enabling unauthorized access, modification, disclosure, or control.
- **Semantic Corruption**: corruption of the intended meaning or structure of data crossing a software interface, such that the receiving component interprets the data in an unintended domain (e.g., as code, commands, object structure, or control input).

These categories operate strictly at the error level. They describe the semantic form of the runtime deviation itself, not the underlying defect, the violated resource constraint, the exploit mechanism, or the externally observed failure.

CWE-to-error mapping. Each CVE is assigned a single error label by mapping a selected primary CWE to one of the categories above. Table A.2 the representative CWE-to-error mappings used in this thesis. Injection-related CWEs are mapped to *Semantic Corruption* because exploitation corrupts the semantic structure of data exchanged across interfaces, causing it to be interpreted in an unintended domain. CWE-116 explicitly captures this notion by noting that the intended structure of a message is not preserved. Deserialization and prototype pollution are likewise mapped to *Semantic Corruption*, as they corrupt object or execution semantics rather than internal memory state. Resource-access CWEs such as path traversal, unrestricted file upload, and

XML external entity resolution are mapped to *Authority Violation*, as the runtime deviation consists in breaching authority boundaries between the component and protected external resources. Memory- and arithmetic-related CWEs are mapped to *State Corruption*, as exploitation produces an invalid or inconsistent internal state, including corrupted memory, invalid pointers, or incorrect numeric values. While some numeric defects may later enable other deviations (e.g., out-of-bounds writes), the error label reflects the observed runtime manifestation.

Table A.2: Mapping from representative CWEs to semantic error categories

CWE ID	CWE Name	Error Category
78	OS Command Injection	Semantic Corruption
89	SQL Injection	Semantic Corruption
94	Code Injection	Semantic Corruption
116	Improper Encoding or Escaping of Output	Semantic Corruption
502	Deserialization of Untrusted Data	Semantic Corruption
1321	Prototype Pollution	Semantic Corruption
1336	Template Injection	Semantic Corruption
22	Path Traversal	Authority Violation
434	Unrestricted File Upload	Authority Violation
611	XML External Entity Reference	Authority Violation
120	Buffer Overflow	State Corruption
416	Use After Free	State Corruption
476	NULL Pointer Dereference	State Corruption
787	Out-of-bounds Write	State Corruption
190	Integer Overflow or Wraparound	State Corruption
191	Integer Underflow	State Corruption
704	Incorrect Type Conversion or Cast	State Corruption

Primary CWE selection rule. When multiple CWEs are listed for a CVE, a single *primary CWE* is selected to anchor error labeling. Selection is guided by the weakness described in the CVE text and the reported runtime behavior. When the nominal CWE reflects an upstream cause but exploitation produces a different concrete manifestation (e.g., an integer overflow leading to a heap out-of-bounds write detected by AddressSanitizer), the error is labeled according to the *observed semantic deviation*. This keeps the labeling aligned with the bug–error–failure model and with the manifestation objective of the taxonomy.

Protocol. Error labeling follows a fixed protocol: (i) identify candidate CWEs from NVD metadata, (ii) analyze the CVE description to determine the runtime deviation, (iii) inspect contextual evidence when needed to resolve ambiguity, and (iv) assign a single semantic error category from the taxonomy above.

A.2.3 Failure taxonomy and impact mapping

This subsection defines the failure taxonomy used to label the *failure* side of reconstructed vulnerability chains and specifies how impact statements from CVE descriptions are mapped into normalized categories. Because CVE impact phrasing varies in form and granularity, these statements are mapped into a small set of failure labels capturing the externally observable consequences of exploitation. The mapping follows recurring patterns shared across established frameworks, leveraging their conceptual overlap to ensure consistent and comparable labeling across cases.

Conceptual basis. The taxonomy is grounded in classical dependability theory, in which a *failure* is defined as a deviation of delivered service from its specified or intended behavior [10]. In the fault–error–failure chain, failures represent the externally visible outcomes that arise when internal errors propagate beyond the system boundary. Unlike bugs and errors, failures in security contexts are rarely standardized at a fine granularity. Dependability theory [127] classifies failures into abstract categories such as content, timing, and halt failures, which are suitable for general system analysis but too coarse for vulnerability characterization. In practice, security failures are usually described through their *technical impact*, that is, what an attacker gains or what service property is violated as a result of exploitation. Multiple standards and frameworks, despite using different terminology, they overlap conceptually and converge on the same notion of impact:

- CVE descriptions explicitly include an impact field indicating what an attacker gains [161].
- CWE captures technical impacts in the `Common_Consequences` element [160].
- CAPEC [159] enumerates common attack consequences such as privilege escalation and code execution.
- CVSS [155] models impact in terms of confidentiality, integrity, and availability.
- STRIDE [158] threat model similarly describes externally visible security failures.
- NIST's Vulnerability Data Ontology [170] and BF [19] also define failures as recognized results of exploitation, such as code execution or information exposure.

Failure categories. Based on this convergence and on empirical analysis of CVE impact descriptions, we adopt a small set of failure categories that reflect the most common externally observable consequences of exploitation. We use the CVE impact field as a starting point, as it maps most directly to observed practice, with minor terminological normalization and extensions. The resulting failure categories are:

- **Code Execution**: the attacker can execute arbitrary or unauthorized code or commands within the vulnerable system.
- **Information Exposure**: the attacker can read, infer, or otherwise obtain sensitive or protected information.

- **Denial of Service**: the attacker can disrupt normal service operation, causing crashes, hangs, or resource exhaustion.
- **Elevation of Privilege**: the attacker can gain privileges or capabilities beyond those initially authorized.
- **Tampering**: the attacker can modify data, state, or system artifacts in unauthorized manners.
- **Bypass**: the attacker can circumvent a protection or enforcement mechanism, such as authentication or authorization checks.

These categories describe failures strictly at the service boundary. They characterize what observable capability, access, or disruption results from exploitation, independently of the internal error that enabled it.

Impact-to-failure mapping. Failures are identified by extracting impact-related phrases from CVE descriptions and labeling them using the failure categories defined above. The mapping is phrase-driven and reflects how impacts are described in practice, focusing on what capability the attacker gains or what service property is violated. Representative mappings include:

- Phrases indicating execution of attacker-controlled code or commands (e.g., "execute arbitrary code," "remote code execution") are labeled *Code Execution*.
- Phrases describing service disruption or crashes (e.g., "denial of service," "cause a crash") are labeled *Denial of Service*.
- Phrases referring to unauthorized disclosure of sensitive data (e.g., "information disclosure," "read internal files") are labeled *Information Exposure*.
- Statements indicating unauthorized privilege gain (e.g., "escalate privileges") are labeled *Elevation of Privilege*.
- Descriptions of unauthorized modification of data or system artifacts (e.g., "write arbitrary files," "memory corruption") are labeled *Tampering*.
- Phrases describing circumvention of protection mechanisms (e.g., "bypass authentication") are labeled *Bypass*.

Multi-consequence handling. A single vulnerability may enable multiple failures (e.g., information exposure followed by code execution). When multiple consequences are described, a single failure label is assigned using the following rule: select the **most severe** failure when severity can be ordered unambiguously. Otherwise, select the **first enabled consequence** in the exploitation chain. This rule avoids overcounting while preserving the dominant observable impact.

Protocol. Failure labeling follows a fixed protocol: (i) extract impact-related phrases from the CVE description, (ii) map each extracted concept to a failure category, and (iii) apply the multi-consequence handling rule to select a single failure label. This process ensures that failure labels reflect observable consequences, consistent with dependability theory and with the manifestation objectives of this thesis.

A.2.4 Software Type Taxonomy for Labeling Vulnerable Applications

This subsection defines the taxonomy used to label the software type of each vulnerable target in the profiling dataset. Its purpose is not to exhaustively characterize software artifacts, but to enable consistent, coarse-grained categorization across a heterogeneous corpus. Each repository or packaged artifact is assigned a single label that captures its *primary deliverable* and *execution context*. The taxonomy reflects a pragmatic distinction used in contemporary software ecosystems between deployable applications and services, reusable components, and host-dependent add-ons. Framing software types at this level supports meaningful comparison across targets that differ substantially in language, scale, and internal architecture, while remaining interpretable to practitioners.

Design rationale. The labels listed in Table A.3 were selected with three criteria in mind. First, they correspond to common and intuitive roles in real-world software production and deployment. Second, they are sufficiently distinct to reduce ambiguity during dataset curation, which is particularly important in a manually labeled corpus. Third, they are meaningful for security profiling, because different software types expose different trust boundaries and dominant input channels, such as externally request-driven services, embedded dependencies consumed by other software, or user-invoked tools. More broadly, the taxonomy adopts an *intended consumption* perspective, labeling software according to how it is meant to be used—deployed, run, imported, or installed—rather than according to implementation language or internal design.

Label assignment. Label assignment follows a deliberately simple, single-type rule grounded in observable project signals. Documentation, installation instructions, and the dominant happy path of usage are used to determine whether a project is primarily intended to be deployed and run, imported or linked as a dependency, or installed into a host platform. When a project spans multiple roles—for example, offering both a command-line interface and a library API—the assigned label reflects the role presented as the main product, while auxiliary artifacts are treated as secondary. This operationalization prioritizes clarity and reproducibility over fine-grained expressiveness, which is appropriate given the taxonomy's role as a supporting dimension in a larger empirical analysis.

Alignment with prior work. To strengthen labeling reliability and external validity, the definitions and assignment decisions were aligned with manually labeled software-type data reported by Ezenwoye et al. [58]. More broadly, this taxonomy is informed by prior work emphasizing the importance of explicit software-type and domain taxonomies for repository-centric analysis,

Type	Description	Scenarios	Disambiguation Rule
Extension	A package installed *into* a host product to add/-modify functionality; not meaningful standalone.	WordPress/Drupal plugins and themes; IDE/editor extensions; browser extensions; add-ons in host UI.	It requires a separate host application to run ("install into X", "enable in X", "drop into plugins/").
Library	Reusable code intended to be imported/linked by other software (API-first, not app-first).	SDKs and client libraries; shared "core" packages; packages installed via dependency managers and consumed via imports.	If the "happy path" is "pip/npm install + import + call", label Library even if it's marketed as a framework.
Framework	A development foundation that provides lifecycle, conventions, and scaffolding to build apps (by inverting control).	Web frameworks; application skeleton generators; middleware pipelines; systems defining project structure and how code is invoked.	If the "happy path" is "create app / conform to structure / register handlers / framework runs and calls you".
Utility	A standalone tool that performs tasks on demand (often CLI), typically runs and exits.	Linters/scanners; converters; build tools; migration scripts; admin tools executed as commands producing artifacts/reports.	The dominant interface is a command that runs to completion (non-daemon); it is not meant to remain running.
Server	A long-running process providing networked functionality (daemon/service) consumed over a network interface.	Web services/APIs; gateways; agents; databases; background services managed by systemd/containers, listening on ports.	It is intended to run persistently and accept network requests/events; it is not meant to be the web UI in a user-facing application.
Client	Software whose primary role is to initiate connections to a remote service/peer and consume it (implements a protocol/API; parses remote responses).	RDP/SSH/VNC clients; chat/messaging clients; game clients; CLI/GUI tools that mainly call remote APIs and present results.	The main runtime role is outbound consumption of a remote service; it is not meant to primarily accepts inbound; it is not a one-shot local tool over files/stdin/args.
Web App	A user-facing application primarily accessed through a browser (often deployed on a web server with a database).	CMS/CRM/ERP/portals; systems installed via "deploy + configure DB + open URL" and used interactively by end users.	The host product users log into via browser; it is not meant to be installed into a host.
Mobile App	An application intended to run on mobile OSes with native packaging and platform-specific build artifacts.	Android/iOS apps distributed via APK/IPA or app stores; projects with Gradle/Xcode and mobile permission models.	The primary runtime target is Android/iOS with native packaging; it is not meant to be accessed mainly via browser.

Table A.3: Software type taxonomy for labeling vulnerable applications in the profiling dataset.

including structured vocabularies for software classification [68], lexicon-based techniques that infer software domains from source code terminology [27], and automated classifiers that categorize repositories at scale by application domain [251]. These studies differ in scope and granularity, often focusing on domain rather than software type. Still, they support the need for clearly defined categories and transparent labeling procedures.

A.2.5 Functional Area Taxonomy for Labeling Vulnerable Software Components

This subsection defines the functional-area taxonomy used to label vulnerable components in the dataset. The objective is to characterize which application features or subsystems are repeatedly implicated by vulnerabilities, enabling susceptibility analysis across software types (Web App, Library, Client, Server, Extension, Utility; see subsection A.2.4).

Functional Area definition. CWE defines `Functional_Area` as the functional area of the software in which a weakness is most likely to occur; a single CWE entry may include multiple functional areas. This study uses CWE functional areas as a controlled vocabulary for application-level subsystem labels, while preserving CWE's intent that the value indicates *where* weaknesses tend to appear [160].

Deriving the value set. Rather than relying on an informal or ad hoc enumeration, functional-area values were obtained programmatically by parsing the official CWE XML and extracting the distinct strings used in `Functional_Area` elements across all CWE entries. This follows CWE's published schema and vocabulary [160], including the `Functional-Area-Independent` special value for weaknesses not limited to a small set of functional areas.

Labeling target. Functional-area labels are assigned from the bug location identified through the vulnerability-fixing patch and the defective code region that enables the vulnerable execution path. The CVE's CWE label is not used to infer functional area, because it often describes the manifestation or error rather than the subsystem containing the enabling bug.

Taxonomy used in this study. Table A.4 documents the functional-area labels used in this study and the higher-level categories used to group them for analysis. For this study, software functional area is operationalized as the application-level feature or subsystem in which the bug resides, as identified from the vulnerability-fixing patch and the corresponding defective code region.

Out-of-scope functional-area values. Some CWE functional-area values are treated as out of scope because the dataset is restricted to open-source application software and excludes operating-system and hardware or firmware concerns. In particular:

- `Power` and `Clock` typically correspond to platform-level power-management and timing concerns, which rarely define stable application feature boundaries in this dataset.
- `Signals` commonly denotes OS-level asynchronous notification or interrupt mechanisms, which fall outside the study scope.

In addition, several application-level functional areas were removed from the target label set because they did not occur in the analyzed dataset: `Session Management`, `Interprocess Communication`, `Counters`, `Cryptography`, and `Dependency/Library Integration`.

Table A.4: Functional-area taxonomy for labeling vulnerable components.

Functional Area	Description	Examples
Authentication	Code responsible for identity verification and login/token issuance flows.	Credential checks; auth middleware/hooks; token issuance endpoints.
Security Enforcement	Code that enforces security policy by gating privileged actions or applying mandatory checks.	Authorization checks; sandbox/guard rails; security filters/wrappers.
String Processing	Code that parses/normalizes/interprets text into structure across trust boundaries.	Encoding/decoding; normalization; templating; XML/JSON parsing; key-path splitting.
Number Processing	Code where numeric semantics affect safety-critical bounds, indices, sizes, or allocations.	Integer overflow/underflow; casts; size computations; array indices/length checks.
File Processing	Code that handles filesystem paths, file formats, and file boundary behaviors.	Path handling; archive extraction; upload/download; symlink/junction issues; permissions at file boundary.
Networking	Code implementing network I/O, protocol parsing/framing, and protocol state handling.	Socket/protocol handlers; message framing; request/response parsing at protocol layer.
Resource Management	Code governing memory safety properties and structured resource access.	Buffer sizing; bounds checks; object lifetime/ownership; UAF/overflow surfaces; DOM tree traversal; database record retrieval; resource lookup.
Logging	Code that generates or processes diagnostic/audit records and related telemetry.	Audit events; diagnostic logs; log injection; sensitive data in logs.
Error Handling	Code that governs failure-mode behavior and error-driven control/data flow.	Exception paths; unsafe fallbacks; ignored error returns; cleanup inconsistencies.
Program Invocation	Code that launches external programs or interpreters or executes embedded scripts.	Command construction; argument quoting; environment inheritance; search-path use; script runner calls.

To preserve labeling precision and reduce class sparsity, only functional areas observed in the dataset are retained as target labels. Removed values can be reintroduced in future samples if explicit criteria are met.

Terminology refinement. Three labels from the CWE functional-area values are refined to better match the code region associated with the root cause in the patch of open-source software.

- `Code Libraries` is rendered as *Dependency/Library Integration* to emphasize that the label applies to the vulnerable project's integration layer—such as calls, wrappers, loaders, or bindings—rather than to the entire upstream library.
- `Protection Mechanism` is rendered as *Security Enforcement* to emphasize enforcement points such as guards, checks, filters, and policy gates as the code locus being labeled.
- `Memory Management` is broadened to *Resource Management* to cover both memory-safety properties (allocation, bounds, lifetime, use-after-free and overflow surfaces) and

structured resource-access patterns such as DOM tree traversal, database record retrieval, and configuration or resource lookup.

Rationale and disambiguation heuristics. When a code region could plausibly map to multiple areas, labeling prioritizes the *primary boundary* crossed by the code, that is, the first major trust or abstraction boundary at which untrusted data is interpreted or applied. It also favors *mechanism over intent*: the subsystem where the risky behavior occurs is labeled rather than the feature's higher-level purpose. To reduce ambiguity, two deterministic tie-breakers are applied:

- **Parser override**: if the root-cause region lies in parser, decoder, serializer, or parser-like configuration logic, label it as *String Processing* for textual structured formats such as XML or JSON, and as FILE PROCESSING for file or archive formats and path-containment logic.
- **Exec override**: if the code constructs or executes OS commands or scripts from inputs, label it as *Program Invocation* even if the surrounding feature appears file-oriented or diagnostic.

Appendix B

Implementation Details (Chapter 3)

This appendix documents the procedures used to construct the benchmark described in Chapter 3. It details how OSS-Fuzz OSV entries are qualified and filtered (Section B.1), how qualified entries are linked to CVEs through multi-signal matching (Section B.2), how vulnerabilities are reconstructed and their fixes verified in historically consistent environments (Section B.3), and how defect locations are inferred and structurally compared to crash sites (Section B.4). These sections make explicit the data sources, transformation steps, and validation rules that underpin the version-anchored, manually verified benchmark instances used in the chapter.

B.1 Metadata Validation for OSS-Fuzz OSV Records

This section operationalizes the metadata validation defined in Section 3.4.1. The procedure comprises three components. Section B.1.1 specifies issue-tracker resolution and traceability requirements. Section B.1.2 formalizes crash-type normalization and manifestation-level CWE mapping. Section B.1.3 defines commit-range validation and repository availability checks.

B.1.1 OSS-Fuzz Issue-Tracker Resolution and Traceability

Reproducing OSS-Fuzz vulnerabilities requires access to the original fuzzing issue report, which contains operational details not preserved in OSV summaries, such as the specific fuzzer target, crash context, and references to triggering artifacts. We therefore require that each OSV record include at least one reference to an OSS-Fuzz issue-tracking system. Valid references are those whose host corresponds to either `bugs.chromium.org` (historical OSS-Fuzz reporting) or `issues.oss-fuzz.com` (current infrastructure), ensuring that each retained entry can be tied to its underlying fuzzing report.

Issue resolution normalizes both host formats to a stable issue identifier. For Chromium-hosted URLs, the page is resolved to extract the embedded redirect target that points to the canonical OSS-Fuzz issue page, from which the issue ID is obtained. For URLs already hosted on

`issues.oss-fuzz.com`, the identifier is parsed directly from the URL structure. An OSV entry is retained only if at least one reference yields a valid normalized issue URL and issue ID under this procedure.

B.1.2 Crash-Type to CWE Mapping (Manifestation-Level)

Motivated by the crash-to-CWE correspondences introduced by Keller et al. [115], we assign CWE identifiers to characterize how vulnerabilities manifest at runtime, since such labels are not provided in OSS-Fuzz reports or OSV records. While Keller et al. established the feasibility of mapping crash types to CWE categories, we adopt a more fine-grained mapping design. Rather than aggregating multiple CWE variants under generalized crash classes, we favor one-to-one correspondences whenever the crash semantics uniquely determine the weakness type. When crash descriptors encode both a structural memory context and an operation, we preserve both dimensions through a secondary category. For example, OSV-2021-304 reports a crash of type `Stack-buffer-overflow WRITE 6`; we assign CWE-787 (Out-of-Bounds Write) to capture the write operation and CWE-121 (Stack-based Buffer Overflow) to capture the stack-specific memory region. Operationally, we extract (i) the *Crash type* string (e.g., buffer overflow, use-after-free), and (ii) the *Crash state* lines from the structured sanitizer output, which provide a compact representation of the observed failure context. Then, the mapping relies on (1) a crash-category to CWE table, (2) an operation-to-CWE table for READ/WRITE refinements, and (3) normalization of crash-type formatting variants, producing a *primary* manifestation CWE and an optional *secondary* operation-specific CWE.

Separately from the mapping procedure itself, we constrain which CWE identifiers may be used. Candidate CWEs were screened against MITRE's Vulnerability Mapping Notes, which annotate entries with recommended mapping statuses. We restrict selections to CWEs designated as *ALLOWED*, excluding overly general or discouraged classes even when they appear in prior mappings; for instance, CWE-119 was excluded due to its overly broad scope. For crash types absent from Keller et al.'s catalog, we performed a bottom-up inspection of the CWE taxonomy to identify the most precise one-to-one correspondence consistent with the crash semantics. The resulting extensions are summarized in Table B.1. Crash types judged too generic or operationally uninformative (`UNKNOWN`, `object-size`, `timeout`, `check-failed`) were excluded to avoid introducing weak or ambiguous manifestation labels.

B.1.3 Range Status and Commit Availability

A benchmark instance must be version-anchored: the defect must have an explicitly bounded lifetime in the codebase, defined by an introducing revision and a fixing revision, and those revisions must remain retrievable. We therefore analyze the `git` ranges provided in OSV records and retain only vulnerabilities for which at least one range includes both an *Introduced* and a *Fixed* event. Records lacking either endpoint are excluded because the defect lifetime cannot be reliably established.

Table B.1: Additional crash-type to CWE mappings introduced beyond Keller et al. [115].

Crash Type	CWE	Rationale
Use-after-return	CWE-562	Access to stack memory after function return corresponds to exposing a stack variable address beyond its lifetime.
SEGV on unk. address	CWE-476	Dereference of an unmapped or zero-page address is a canonical NULL pointer dereference.
Security exception	CWE-248	Unhandled exceptions propagating to program termination match uncaught exception conditions.
negative-size-param	CWE-195	Negative size arguments interpreted as large unsigned values reflect signed-to-unsigned conversion errors.
invalid-free / bad-free	CWE-672	Unsafe frees after resource release, including double or invalid frees, align with operations on expired resources.
use-after-poison	CWE-416	Access to poisoned memory regions corresponds to use-after-free behavior.
stack-use-after-scope	CWE-562	References to stack variables retained beyond scope match exposure of stack variable addresses.
memcpy-param-overlap	CWE-475	Overlapping regions in `memcpy` invoke undefined behavior for API inputs.
Out-of-memory	CWE-770	Memory exhaustion due to unbounded allocation reflects allocation of resources without limits.
Incorrect-function-pointer-type	CWE-686	Calling through a function pointer with an incompatible signature matches incorrect argument type usage.

Beyond the presence of endpoints, we verify the availability of the underlying historical artifacts. For each candidate range, the referenced repository must be resolvable and both boundary commits must be retrievable; records pointing to missing repositories or non-resolvable commits are excluded. When multiple introduced or fixed events are specified (e.g., across branches or backports), we select a representative boundary by comparing commit timestamps, choosing the oldest introduction and the newest fix to define a consistent defect interval. This selection provides a single, stable introduced–fixed pair for downstream reconstruction without attempting to model multi-branch maintenance behavior.

B.2 Operational Details of OSS-Fuzz–to–CVE Linkage

This section operationalizes the OSS-Fuzz–to–CVE linkage workflow defined in Section 3.4.2. The procedure comprises four components. Section B.2.1 defines product alignment through repository–product normalization. Section B.2.2 specifies weakness alignment via CWE intersection. Section B.2.3 describes reference-level provenance matching using issue and commit links. Section B.2.4 formalizes semantic disambiguation and priority-based match selection.

B.2.1 Product Alignment and Repository–Product Normalization

The first stage of OSS-Fuzz–to–CVE linkage establishes a shared product namespace between fuzzing-derived vulnerability records and CVE configurations. OSS-Fuzz vulnerabilities are organized around source-code repositories, whereas CVEs describe affected software using vendor–product identifiers derived from CPE-style taxonomies. Product alignment therefore translates repository identities into the product identifiers used in CVE records so that both datasets refer to the same software entities.

Repository-to-Product Mapping. The starting point is the repository-to-product resolution produced in the previous chapter (Appendix A.1.6), where each application entry in the CPE dictionary was mapped to a corresponding GitHub repository. That mapping associates repository identifiers with one or more canonical `vendor:product` strings and serves as the primary source of product normalization for OSS-Fuzz entries during CVE linkage. Certain OSS-Fuzz projects are not covered by the earlier resolution process—typically due to naming discrepancies or the absence of a direct CPE entry. In such cases, the mapping is supplemented with a curated set of repository-to-product overrides to ensure that frequently analyzed OSS-Fuzz targets absent from automated resolution still participate in product alignment.

Alignment and Filtering Criteria. Each OSS-Fuzz metadata entry inherits the list of product identifiers associated with its repository. Entries for which no repository-to-product mapping can be established are excluded from CVE matching, since product alignment forms the gateway condition for all subsequent stages. The union of resolved product identifiers constitutes the product pool against which CVE configuration data are compared. For each CVE, vulnerable products listed in its configurations are normalized to the same `vendor:product` format, and a candidate OSS-Fuzz match is considered only if the two product sets intersect. This stage filters the metadata-qualified population to entries that can be expressed in the same product vocabulary as CVE records, ensuring that later weakness and reference comparisons operate only on vulnerabilities referring to the same software systems.

B.2.2 Weakness Alignment via CWE Intersection

Following product alignment, the second linkage stage requires agreement regarding the *type* of vulnerability described in each source. OSS-Fuzz entries are characterized at the manifestation level through crash-derived CWE identifiers (Appendix B.1.2), whereas CVE records include one or more CWE classifications supplied during vulnerability analysis. Weakness alignment therefore establishes whether the two records describe compatible classes of software flaws, providing a taxonomy-level consistency check before considering more specific evidence.

CWE Normalization and Comparison. For OSS-Fuzz entries, the normalized manifestation-level CWE identifiers derived during extraction are used, consisting of a primary CWE capturing

the core weakness category and, where applicable, a secondary CWE refining the operation or memory context. These identifiers are reduced to their numeric form to enable direct comparison. For CVEs, all CWE identifiers listed in the NVD weakness fields are collected. A candidate OSS-Fuzz–to–CVE pair passes this stage only if at least one of the OSS-Fuzz CWEs (primary or secondary) intersects with the set of CWEs associated with the CVE.

Compatibility Filtering and Scope. The intersection requirement filters cases where product overlap exists but the two records describe fundamentally different vulnerability classes (e.g., a memory-safety violation versus a logic or configuration error). Because CWE annotations in CVEs may vary in granularity, this stage functions as a compatibility check. It ensures that pairings forwarded to later stages share a coherent weakness taxonomy. Entries failing to satisfy this criterion are excluded from further consideration, constraining subsequent provenance matching to vulnerabilities consistent in both software context and flaw type.

B.2.3 Reference-Level Provenance Matching

After product and weakness compatibility are established, the third linkage stage seeks direct evidence of provenance to indicate that an OSS-Fuzz entry and a CVE refer to the same underlying defect. Unlike prior stages, which rely on structural descriptors, this step evaluates artifacts embedded in CVE reference fields that point to concrete development or reporting resources. Such references frequently originate from disclosure discussions or patch documentation and therefore provide stronger signals of shared identity across datasets.

Issue-Level Provenance Alignment. The first category of reference evidence consists of OSS-Fuzz issue-tracker links. CVE records often include external references to bug reports or discussion threads; when these URLs resolve to OSS-Fuzz issue-tracking systems (either historical Chromium-hosted reports or the current OSS-Fuzz tracker), the corresponding issue identifiers are extracted. These identifiers are compared against the normalized OSS-Fuzz issue IDs associated with each candidate entry (Appendix B.1.1). A match indicates that both records reference the same fuzzing report and constitutes high-confidence provenance alignment.

Commit-Level Provenance and Filtering. The second category consists of commit-level references. Some CVEs include links to source-code commits documenting the introduction or remediation of the vulnerability. Commit hashes extracted from such URLs are compared against the introduced and fixed revisions recorded in the OSS-Fuzz metadata (Appendix B.1.3). Overlap at this level provides independent historical linkage by tying both records to the same code change. Duplicate URLs are ignored during processing to prevent repeated evidence from inflating confidence. Candidate pairs exhibiting at least one issue-level or commit-level match pass this stage and are considered to have explicit cross-source provenance signals. Pairs lacking such evidence are retained only for subsequent semantic disambiguation, ensuring that reference-level evidence is prioritized over structural similarity alone.

B.2.4 Semantic Disambiguation and Priority-Based Match Selection

Even after product alignment, weakness compatibility, and reference-level provenance matching, ambiguity can remain. A single CVE may correspond to multiple OSS-Fuzz entries that satisfy structural and historical criteria, particularly when vulnerabilities occur within the same project and share similar weakness types. The final linkage stage performs semantic disambiguation and applies a priority-based selection policy to ensure that each retained correspondence reflects the strongest available combination of evidence.

Semantic Disambiguation via Program-Entity Cues. Semantic cues are derived from location-specific information contained in CVE descriptions and OSS-Fuzz crash summaries. Crash reports include final stack frames at the failure point, providing function names and source file paths near the crash site. CVE descriptions are scanned for tokens referring to program entities (e.g., function or file names), and overlap with identifiers extracted from crash frames is used as a weak signal that both records reference failures in the same code region. For example, the crash state of OSV-2023-270 includes the function `bit_utf8_to_TU`, and CVE-2023-36272 describes a heap buffer overflow in that same function at `bits.c`. Another CVE referencing the same commit (CVE-2023-36271) instead mentions `bit_wcs2nlen`, enabling keyword-based differentiation between otherwise structurally similar candidates. These cues are applied only when a stronger product, weakness, or reference evidence does not uniquely determine the correspondence.

Priority Ladder and Consolidation Policy. Match selection follows a priority ladder that ranks evidence combinations according to evidentiary strength. The highest priority is assigned when reference-level evidence (issue or commit overlap) is reinforced by semantic agreement. The next level corresponds to cases where issue-level and commit-level evidence intersect directly, indicating convergence between reporting and code-history artifacts. Lower priority is assigned when only one form of reference evidence is present, and the lowest priority corresponds to matches relying primarily on semantic similarity after structural alignment. Each candidate OSS-Fuzz entry is assigned a match-priority value reflecting the strongest evidence combination. When multiple candidates remain for the same OSS-Fuzz identifier, only those with the best (lowest) priority value are retained. If more than one candidate shares the top priority and cannot be uniquely distinguished, the correspondence is discarded to avoid introducing uncertainty into the mapping. This consolidation step ensures that the final OSS-Fuzz–to–CVE mappings represent unique, evidence-weighted correspondences rather than heuristic associations.

B.3 Automated Reproduction and Fix Verification Procedure

This section operationalizes the reproduction and fix-verification workflow defined in Section 3.4.3. The workflow comprises four components. Section B.3.1 defines the execution architecture. Section B.3.2 specifies context reconstruction, including issue retrieval, snapshot resolution, and

repository initialization. Section B.3.3 describes containerized build and execution under OSS-Fuzz infrastructure. Section B.3.4 formalizes crash parsing and structural equivalence checking.

B.3.1 Tooling Architecture and Execution Model

This subsection describes the operational structure of the automated reproduction workflow referenced in Section 3.4.3. The workflow is implemented in a publicly available command-line tool, `osv-reproducer`[1], which orchestrates repository resolution, environment reconstruction, build execution, crash replay, and fix verification for individual OSS-Fuzz vulnerabilities identified by OSV IDs. The tool operates deterministically on one vulnerability at a time and produces structured output artifacts documenting reproduction status, execution logs, and verification outcomes. Its purpose is not to discover vulnerabilities, but to reconstruct historically grounded execution states and validate that a documented crash can be reproduced and subsequently eliminated by the recorded fixing revision.

Reproduction and Verification Modes. The tool exposes two complementary execution modes corresponding to the conceptual workflows defined in the main methodology. In *crash reproduction mode*, the pipeline reconstructs the introducing revision of the project and attempts to trigger the documented failure using the original OSS-Fuzz test case. In *fix verification mode*, it reconstructs the version containing the recorded fixing commit and re-executes the same test case to confirm the absence of the crash. The two execution modes share a common architectural backbone and differ only in the targeted revision and the expected validation outcome (presence versus absence of the crash), while the surrounding workflow remains structurally identical.

Architectural Structure and Execution Workflow. Although earlier stages validate that an OSS-Fuzz entry is eligible for reproduction, the reproduction workflow does not directly consume their metadata. Execution proceeds through a fixed sequence of stages: **context resolution**, **project build under OSS-Fuzz–compatible instrumentation**, **test-case execution**, and **structured validation of outcomes**. The initial stage produces a structured *reproduction context* that encapsulates the vulnerability identifier, execution mode, issue-report metadata, project configuration, repository boundaries, and the timestamp anchoring the historical snapshot. This context serves as the execution anchor for the subsequent build, execution, and validation stages. Each stage produces a structured status outcome, allowing failures to be attributed precisely to missing context, build instability, execution divergence, or verification mismatch.

B.3.2 Context Reconstruction and Snapshot Resolution

This subsection details how an execution context is reconstructed for a given OSV identifier. Reproduction requires more than commit boundaries: it depends on the original OSS-Fuzz issue report, the crash metadata embedded within that report, the corresponding build snapshot, and the

[1] https://pypi.org/project/osv-reproducer/

exact repository revisions referenced at the time of failure. The purpose of context reconstruction is to assemble a self-contained *reproduction context* that binds issue-report semantics, snapshot state, and repository boundaries into a single, temporally coherent execution anchor.

Issue Retrieval and Snapshot Resolution. Context reconstruction begins by resolving the OSS-Fuzz issue identifier associated with the OSV record. The issue report is fetched from the OSS-Fuzz issue tracker (e.g., `https://issues.oss-fuzz.com`) and parsed into a structured representation. During parsing, legacy field names are normalized, and crash-specific fields are consolidated into a dedicated *CrashInfo* object that captures, at minimum, the reported impact category and structured stack information. Reports lacking sufficient crash descriptors are rejected at this stage. Snapshot resolution determines the OSS-Fuzz build state corresponding to the observed crash. If a timestamp has previously been recorded for the OSV identifier, the corresponding snapshot is loaded directly. Otherwise, a timestamp is inferred from either (i) the regression range encoded in the issue report (for crash reproduction mode) or (ii) the temporal window derived from the fixing revision (for verification mode). Snapshots are retrieved from the OSS-Fuzz archival storage in the Google Cloud Storage bucket `clusterfuzz-builds`. When an exact timestamp is known, the corresponding snapshot file is downloaded directly; otherwise, available snapshots within the inclusive time interval are enumerated, and the most recent valid match is selected. Failure to obtain a valid snapshot halts context reconstruction.

Project Configuration and Repository Initialization. Once a snapshot is obtained, the repository descriptors embedded in it are used to reconstruct the project's source state. Each entry specifies a remote repository location and a pinned revision. Only repositories hosted on GitHub are supported; each repository is cloned at the selected commit to establish a local version. If cloning fails or no valid repositories can be initialized from the snapshot, context reconstruction terminates with an error. In parallel, a version of the OSS-Fuzz project configuration up to the snapshot's timestamp is retrieved. This includes the Dockerfile for the project and associated build resources. The Dockerfile is parsed to identify required mountable sources and downloadable build inputs, ensuring that all files referenced by build instructions are available locally. The issue report (including its normalized *CrashInfo* and triggering test case ID), the resolved snapshot timestamp, the initialized repositories, and the project configuration are then assembled into the structured *reproduction context*. This context object binds the OSV identifier, execution mode, crash metadata, repository boundaries, and snapshot state into a single, consistent execution anchor for the build and execution stages described previously.

B.3.3 Containerized Build and Instrumentation Reproduction

This subsection describes how projects are built and executed inside Docker containers using the OSS-Fuzz infrastructure. Reproduction requires not only the correct source revision, but also the

correct instrumentation configuration (fuzzing engine, sanitizer, and architecture). To preserve fidelity with the original fuzzing environment, both compilation and execution are performed inside containers configured to mirror OSS-Fuzz's build and runtime assumptions.

Building Fuzzer Container. For each reproduction context, a project-specific base image is constructed from the OSS-Fuzz project directory corresponding to the resolved snapshot. The resulting image is tagged with `osv-reproducer/{project}-{oss_fuzz_repo_sha}:latest`. If the image already exists locally, it is reused; otherwise, it is built directly from the project's OSS-Fuzz configuration. A container is then launched from this image to compile the project with OSS-Fuzz–compatible instrumentation. The container environment is configured using explicit variables, including `FUZZING_ENGINE`, `FUZZING_LANGUAGE`, `SANITIZER`, `ARCHITECTURE`, and `PROJECT_NAME`. The execution platform is selected according to the issue report architecture (e.g., `linux/amd64` or `linux/arm64`). All resolved repositories are mounted into the container at their snapshot-defined paths, and required build-context files identified from the project Dockerfile are mounted accordingly. The container build process produces instrumented fuzzing binaries under `/out`. Build logs are streamed and persisted. If the container exits with a non-zero status or an error code is detected in the logs, the build stage fails, and reproduction does not proceed.

Executing Fuzzer with the Crashing Input. After a successful build, the fuzzer binary is executed inside the OSS-Fuzz runner image `gcr.io/oss-fuzz-base/base-runner:latest`. The previously generated output directory is mounted at `/out`, and the original crashing test case is mounted read-only at `/testcase`. The runner container is launched with `"reproduce {fuzz_target} -runs=1"`. The environment includes `SANITIZER`, `ARCHITECTURE`, and a helper flag enabling structured output capture. The expected exit code depends on execution mode: in crash reproduction mode, exit code `1` indicates a crash; in fix verification mode, exit code `0` indicates successful termination without triggering the fault. Container logs are streamed and parsed to extract structured crash information. When a crash occurs, logs are converted into a normalized *CrashInfo* representation for subsequent structural comparison. If execution does not match the expected outcome for the given mode, the run is classified as a reproduction or verification failure.

B.3.4 Crash Signature Parsing and Matching

This subsection specifies how runtime sanitizer output is converted into a structured crash representation and compared against the crash metadata recorded in the OSS-Fuzz issue report. Reproduction is considered valid only if the observed crash matches the reported failure. Matching operates on normalized fields rather than raw log text, ensuring comparisons are independent of formatting differences.

Parsing Crash into Structured Representation. Runner container logs are scanned for sanitizer markers indicating the start of a failure (e.g., lines beginning with `==...==ERROR`). From

this point, the parser extracts the reported impact category (e.g., `heap-buffer-overflow`, `SEGV`), the memory operation type (`READ` or `WRITE`), the access size (when available), and the faulting address. The primary stack trace is collected until the sanitizer summary marker (`"SUMMARY:"`) is encountered. Each stack frame is normalized into a structured representation that includes the instruction address, the logical function name, and, when present, the associated source location. These frames are assembled into a *CrashInfo* object mirroring the structure produced when parsing the original OSS-Fuzz issue report. If no valid error start marker or stack trace is detected, the run is treated as non-crashing for structural comparison purposes.

Matching Against the OSS-Fuzz Report. For crash reproduction mode, the parsed *CrashInfo* is compared against the issue report's recorded crash metadata. Basic fields are checked first: the impact category and memory operation must match exactly. Size and address comparisons may be performed when explicitly required. If these primary fields diverge, reproduction is rejected. Stack comparison then evaluates frame-level consistency. Because sanitizer instrumentation may introduce additional frames (e.g., wrapper or memory-checking routines), a stack shift procedure is applied: if the first reported frame does not match the first reproduced frame, subsequent frames are examined to determine whether alignment can be achieved by shifting the reproduced stack. If alignment is found, comparison proceeds using the computed offset. At least one frame must match between the report and the reproduced stack; otherwise, it is considered a structural mismatch. In the main text, runs with a structurally matching crash are referred to as *matched* entries, runs that produce a crash failing these criteria are labeled *mismatch*, and runs without a crash are labeled *no crash*. In fix verification mode, the expected condition is that there is no crash. If a structured *CrashInfo* is produced during verification, the patch is considered ineffective, and verification fails. Conversely, in reproduction mode, no crash is treated as a reproduction failure.

B.4 Patch-Driven Root-Cause Localization and Crash Separation

Defect-level ground truth is derived from reproduced vulnerabilities by combining sanitizer stack traces and fixing commits. Localization first extracts the crash-context file–line anchors and aligns them with fixing revisions to obtain a file-scoped diff (Appendix B.4.1). Candidate edit locations are then ranked using a line-distance heuristic and manually validated to determine the bug site (Appendix B.4.2). Finally, the separation between the bug and crash sites is computed from observed call stacks (Appendix B.4.3). Together, these steps yield version-anchored, manually verified bug locations and a quantitative measure of their distance from failure sites.

B.4.1 Crash-Context Anchor Extraction and Snapshot-to-Commit Patch Retrieval

Root-cause localization operates on the structured artifacts produced by the automated reproduction workflow (Appendix B.3). The procedure integrates two complementary evidence streams:

(i) **crash-context anchors** derived from structured sanitizer output, and (ii) **patch evidence** obtained by resolving reproduction snapshots to concrete Git commits and computing their version-controlled differences. These artifacts reflect the exact program states in which the failure manifested and was subsequently corrected. Together, they define a *bounded and version-anchored search space* for defect inference.

Crash-Context Anchor Extraction from Stack Frames. Crash anchors are derived from the structured *CrashInfo* representation emitted during reproduction and persisted for each OSV identifier (Appendix B.3.4). The workflow loads the stored crash metadata together with its associated reproduction context and iterates over the primary stack frames recorded at failure time. For each frame, the `module` field is parsed to extract a *source-relative file path* and *line number*. Frames lacking a valid path–line encoding are discarded. Candidate paths are resolved against the reconstructed project directory corresponding to the crash snapshot (Appendix B.3.2), normalized to source-relative form, and validated against the filesystem state at that revision. Only entries whose paths exist within the project tree are retained. The result is a mapping $\texttt{file} \rightarrow \{\texttt{line}_1, \texttt{line}_2, \ldots\}$ representing empirically validated manifestation points. These **file–line anchors** constitute the runtime evidence that constrains subsequent patch alignment.

Commit Resolution and File-Scoped Diff Retrieval. Patch evidence is derived by resolving both the *crash snapshot* and the *fix snapshot*—materialized during reproduction—to their corresponding Git commits (Appendix B.3.2). Each snapshot descriptor encodes a repository URL and revision identifier; these are converted into concrete repository and commit instances through normalized URL parsing and commit lookup. Let C_{crash} denote the crash-inducing commit and C_{fix} the fixing commit. A diff is computed as $\Delta = \texttt{diff}(C_{\text{crash}}, C_{\text{fix}})$ and is explicitly restricted to the set of files implicated by the crash anchors. This **file-scoped diff restriction** ensures that only developer modifications intersecting runtime-relevant files are considered. If Δ contains no modifications affecting the crash-implicated files, localization is terminated for that entry. The resulting structured diff—composed of normalized patch hunks for the candidate files—serves as the input to the structural proximity analysis described in the subsequent subsection.

B.4.2 Diff-Based Root-Cause Selection via Line Proximity

Given the file-scoped diff Δ between the crash-inducing commit C_{crash} and the fixing commit C_{fix} (Appendix B.4.1), localization identifies the modification that lies closest to the empirically validated crash anchors. The crash site itself is not assumed to be the defect origin; instead, the fixing commit is examined to determine where developers intervened relative to the observed failure. The guiding assumption is pragmatic: edits introduced in the fixing revision are typically located near the code responsible for the crash, even if the defect is not at the exact top stack frame. Line proximity is therefore used as a simple and consistent heuristic to infer the most plausible defect location.

Distance Computation and Candidate Selection. For each patch hunk in Δ, both *deletions* (old lines) and *insertions* (new lines) are compared against the set of crash line numbers for the same file. Let L_{crash} denote the crash line numbers and L_{diff} the line numbers modified within a hunk. The distance is computed as $d = \min_{\ell_c \in L_{\text{crash}}, \ell_d \in L_{\text{diff}}} |\ell_d - \ell_c|$. For each candidate line, the algorithm records a tuple $(\texttt{file_path}, \texttt{lineno}, d, p, t)$, where d is the minimum line distance, p is the *file priority* (derived from the order in which crash files appear), and $t \in \{\texttt{old}, \texttt{new}\}$ indicates whether the line comes from a deletion or an insertion. Selection favors smaller distances; ties are resolved by preferring files with lower priority values. Inserted and deleted lines are treated symmetrically, with their type retained for traceability. The resulting tuple defines the single defect location selected for the vulnerability.

Heuristic Limitations and Manual Verification. The line-distance procedure serves as a heuristic to suggest plausible defect locations rather than to automatically establish ground truth. When the fixing commit does not modify crash-implicated files, or when no candidate line yields a finite distance, the algorithm cannot propose a meaningful location under the proximity criterion. These cases reflect limitations of line-based comparison rather than the absence of a defect. No vulnerability is accepted or rejected solely on the basis of this automated ranking. The selected location is treated as a proposal that guides subsequent inspection. Each vulnerability undergoes manual review of the fixing commit, surrounding context, and crash evidence to confirm or adjust the suggested site. This semi-automatic process ensures that final benchmark entries are validated by expert judgment rather than solely by heuristic proximity.

B.4.3 Call-Graph Separation Between Bug Location and Error Location

Call-graph separation is computed as a post-hoc structural measurement between the manually validated bug location (Section B.4.2) and the error location derived from structured sanitizer output (Appendix B.3.4). The bug function is determined after diff-based candidate selection and manual verification (Appendix B.4.2), while the error function is extracted from the reproduced crash context (Appendix B.4.1). No additional instrumentation, fuzzer arguments, or recompilation steps are introduced at this stage. The metric relies exclusively on the sanitizer-reported stack traces already captured during reproduction. Separation is defined as a count of *call edges* (caller–callee links), not raw stack-frame counts, and therefore approximates dynamic call-graph distance along observed execution paths.

Stack-Based Distance via Lowest Common Ancestor (LCA). Let f_e denote the error function (the highest relevant project frame at the crash site) and f_b the manually labeled bug function. When sanitizer output provides stack traces that include both functions (possibly from different traces, e.g., the crash stack and the provenance stack), separation is computed using a lowest common ancestor (LCA) formulation over the dynamic call paths. Let f_L be the lowest function appearing on both upward traversals of the respective stacks. Let $d(x, y)$ denote the number of

caller–callee edges between functions x and y along a stack segment. The separation is defined as

$$D(f_b, f_e) \;=\; d(f_b, f_L) \;+\; d(f_e, f_L).$$

Operationally, $d(\cdot,\cdot)$ is computed by counting caller transitions between the two functions on the observed trace. This definition yields a single scalar distance even when f_b and f_e are drawn from different sanitizer traces, provided a shared ancestor exists. In fuzzing contexts, this ancestor is frequently the harness entry point (e.g., `LLVMFuzzerTestOneInput`).

Default Distance Under Incomplete Stack Information. Some sanitizer reports—such as segmentation faults or wild-pointer frees—expose only a crash stack and no provenance trace referencing the bug function. In these cases, an LCA-based computation is not possible using sanitizer output alone. To preserve a total metric under this constraint, a conservative default is applied. If `LLVMFuzzerTestOneInput` appears in the crash stack, the crash-to-entry distance $d(f_e, \texttt{LLVMFuzzerTestOneInput})$ (in call edges) is computed, and separation is set to

$$D(f_b, f_e) \;=\; d(f_e, \texttt{LLVMFuzzerTestOneInput}) + 1.$$

The additive constant encodes that the labeled bug function lies outside the observable crash-path segment in the available traces, yielding a consistent lower-bound estimation. If the harness entry function is absent from the crash stack, separation is not computed under this rule.

Separation Categories and Within-Function Line Distance. Based on call-graph distance and file information, each bug–error pair is classified into one of three categories:

- *same function*: bug and error locations are in the same function;
- *different functions, same file*: bug and error are in different functions but in the same file;
- *different files*: bug and error are in different source files.

When bug and error locations are in the same function (call-graph distance 0), line-level separation is additionally measured using within-function line distance: $\Delta = |line_{bug} - line_{error}|$, where $line_{bug}$ and $line_{error}$ are the source line numbers of the bug and error locations, respectively.

Appendix C

Implementation Details (Chapter 4)

This appendix documents the concrete implementation underlying the methodology of Chapter 4. It specifies tool setup and scan orchestration (Section C.1), construction of the root-cause–aligned training dataset (Section C.2), reasoning-layer explanation generation (Section C.3), supervised triage and SARIF-level filtering (Section C.4), and the detailed evaluation protocol for detection and assessment (Section C.5).

C.1 Tool Setup and Scan Pipeline

This section specifies the setup and execution procedures supporting the SAST evaluation described in Section 4.1.2. Section C.1.1 details benchmark dataset normalization, Section C.1.2 documents containerized execution and reproducibility controls, and Section C.1.3 describes tool invocation, input scoping, and SARIF post-processing.

C.1.1 Benchmark / Evaluation Dataset Normalization

To align the scan pipeline with the causally grounded oracle structure introduced in Chapter 3, the dataset is normalized into two evaluation views: *bug-site* anchors (defect/root cause) and *error-site* anchors (manifestation/crash context). Both views share common identifiers (project, CVE/OSV IDs, and commit SHA) but differ in the anchored file/function/line and in the semantics of the associated CWE/operation fields.

Operationally, the pipeline materializes these two views as separate, consistently formatted tables that can be consumed uniformly by the downstream scanner and evaluation logic (e.g., grouping by project/version and extracting a file-level scan scope). The concrete column projections, renaming rules, and file naming conventions are intentionally omitted here to avoid duplicating low-level script details; the complete normalization procedure is provided in the accompanying implementation script.[1]

[1] https://github.com/epicosy/phd_thesis/blob/main/scripts/chapter_4/normalize_data.py

C.1.2 Execution Environment and Reproducibility Controls

Containerized execution. All analyzers are executed inside Docker containers to isolate system dependencies and reduce host-induced variability. The scan runner maps each tool to a corresponding container image and executes analyses over a read-only mount of the repository snapshot (`/src`) with a writable mount for SARIF output (`/out`). This setup ensures a uniform filesystem layout and consistent interaction model across tools without requiring project-specific build environments.

Runtime consistency checks. The runner monitors container execution and verifies that a SARIF artifact is successfully produced before the results are admitted into the evaluation corpus. This provides a uniform failure mode across analyzers and prevents incomplete or failed runs from contaminating downstream processing. Additional low-level runtime preferences (e.g., mount conventions or resource knobs) are treated strictly as execution conveniences and are not methodologically significant.

C.1.3 Tool Invocation and Input/Output Handling

All analyzers are executed using their standard analysis mode and rule coverage consistent with the methodological constraint of evaluating out-of-the-box SAST behavior rather than rule engineering. Listing C.1 shows the template command for *Flawfinder* (v2.0.19), executed from the container image hosted at `003371dbcc47`. The flag `-minlevel=5` restricts output to higher-risk warnings; empirically, varying the threshold between levels 3–5 did not materially change the set of actionable findings while substantially reducing warning volume compared to lower levels. Additional flags (`-falsepositive`, `-neverignore`, `-dataonly`) reduce heuristic suppression and enforce consistent SARIF-oriented reporting.

```
$ flawfinder --sarif --falsepositive --neverignore --quiet --dataonly
  ↪ --minlevel=5 <targets> > /out/<sarif>
```

Listing C.1: Flawfinder command template (v2.0.19).

Listing C.2 shows the template command for *Cppcheck* (v2.18 dev), executed from the container image hosted at `00e1f792e45a`. We set `-max-ctu-depth=9` to match the maximum observed call-graph separation between bug and error anchors in the benchmark, preventing premature truncation of cross-translation-unit reasoning below the dataset's required depth. The remaining parameters enable broad default checking (`-enable=all`, `-inconclusive`) while producing SARIF output.

```
$ cppcheck -j4 --enable=all --inconclusive --output-format=sarif
  ↪ --platform=native --max-ctu-depth=9
  ↪ --output-file=/out/<sarif> <targets>
```

Listing C.2: Cppcheck command template (v2.18 dev).

Listing C.3 shows the template command for *CodeQL* (v2.23.7), executed from the container image hosted at `bb1690789c26`. To satisfy build independence, the database is created in no-build mode (`-build-mode=none`). Analysis uses a curated query suite focused on CWE-relevant defect mechanisms, derived from the official C/C++ security packs but with telemetry, logging, performance, and cryptography-oriented queries removed as benchmark-irrelevant; the exact suite used in the experiments is version-controlled with the thesis artifacts.[2]

```
$ codeql database create <db> --language=cpp --source-root=/src/<root>
  ↪ --build-mode=none --threads=2 --ram=6144 --overwrite
$ codeql database analyze <db> /config/codeql/cwe-focused-cpp.qls
  ↪ --format=sarifv2.1.0 --output=/out/<sarif>
```

Listing C.3: CodeQL command template (v2.23.7).

Listing C.4 shows the template command for *Semgrep* (v1.148.0), executed from the container image hosted at `e60a680f5dd3`. The ruleset is fixed (not tuned per project) and aggregates a local copy of the official C security and correctness rules[3] together with an additional community-provided ruleset to compensate for the limited coverage of the official C pack.[4] The combined rules used in the experiments are archived alongside the thesis configuration.[5] Finally, `-no-git-ignore` ensures benchmark-relevant files are not skipped due to repository ignore patterns, improving consistency across projects.

```
$ semgrep scan --config=/config/semgrep/rules --sarif --no-git-ignore
  ↪ --verbose --sarif-output=/out/<sarif> <targets>
```

Listing C.4: Semgrep command template (v1.148.0).

Input handling. To keep tool outputs proportional with the benchmark oracle and prevent evaluation from being dominated by project-wide reporting volume, analysis is restricted to files linked to benchmark anchors. For each repository snapshot (`owner/repo@commit_sha`), the pipeline derives a set of repository-relative file paths from the evaluation dataset and supplies them as explicit scan targets where supported; otherwise, scanning falls back to the smallest common directory subtree containing those files.

Output handling. Before evaluation, SARIF artifacts undergo normalization and post-processing to remove environment-specific noise while preserving semantic content. Absolute mount prefixes (e.g., `/src/`) are stripped from location URIs to produce repository-relative paths consistent with the scoped input files; non-essential driver metadata, notifications, and invocation records are removed to reduce structural variability across tools; and tool identifiers are standardized where

[2] https://github.com/epicosy/phd_thesis/blob/main/config/chapter_4/codeql/cwe-focused-cpp.qls

[3] https://github.com/semgrep/semgrep-rules/tree/develop/c/lang

[4] https://github.com/0xdea/semgrep-rules

[5] https://github.com/epicosy/phd_thesis/tree/main/config/chapter_4/semgrep/rules

necessary. During post-processing, a finding is considered in-scope only if its *primary SARIF location* corresponds to one of the scoped file paths derived during input handling. This ensures that inclusion checks and downstream metrics operate on consistent location representations without altering the findings themselves, and that evaluation reflects performance strictly within the benchmark-defined observable search space.

C.2 Training Dataset Construction Pipeline

This section documents the procedures used to construct the root-cause–aligned training dataset described in Section 4.4.2. Section C.2.1 details CVE selection and fix-commit constraints, Section C.2.2 specifies diff processing and bug–fix span extraction, and Section C.2.3 describes SAST execution and location-based labeling of warnings.

C.2.1 CVE Selection and Patch Constraints

Training instances are derived from real-world vulnerabilities, but only a subset of CVEs are suitable for learning root-cause–level behavior. The construction procedure therefore enforces two methodological principles: selected vulnerabilities must correspond to well-defined memory-safety defects, and their associated fixes must admit compact, source-localized changes that enable precise bug-site extraction. Rather than modeling coarse-grained security revisions or refactor-heavy updates, the resulting dataset emphasizes semantically grounded defects whose fixes reflect minimal causal corrections.

CVE Corpus Delimitation. Candidate vulnerabilities are drawn from the NVD corpus (Appendix A.1), which provides standardized metadata describing weakness class, affected products, and external references. From this corpus, constraints progressively narrow the set to CVEs that are semantically aligned with root-cause memory-safety defects, grounded in open-source code, and linked to verifiable fixing revisions. Selection is restricted by CWE semantics: only CVEs whose weakness metadata intersects the identifiers `125`, `415`, `416`, `476`, or `787` are retained; these families align with the benchmark's observed failures. To preserve strict train–test independence, any CVE identifier appearing in the evaluation dataset is removed at this stage, preventing both explicit overlap and latent leakage through shared vulnerability instances. Each remaining CVE must reference at least one vulnerable product that can be resolved to a source repository within the curated OSS product catalog (Chapter 2). Resolution proceeds by matching vendor–product identifiers to repository metadata (owner, repository name, and repository ID), ensuring that vulnerability records correspond to version-controlled source artifacts rather than abstract advisories.

Fix Commit Selection. For each admissible CVE–repository pair, candidate commits are treated as potential fix patches and evaluated under structural constraints that favor compact, source-localized defect repairs. When multiple commits are associated with the same vulnerability, only

the *most recent* commit (based on committer timestamp) is retained to reduce inclusion of intermediate or partial fixes superseded by later revisions. Each candidate must have a single parent (merge commits are excluded) so that the vulnerable predecessor revision is unambiguously defined, and must modify exactly one file, constraining the dataset toward patches likely to correspond to a single defect site rather than multi-file refactorings or broad code cleanups. Modifications are restricted to C/C++ source artifacts, requiring at least one changed file with extension `.c`, `.h`, `.cpp`, `.cc`, `.cxx`, `.hpp`, `.hh`, or `.hxx`. For each commit satisfying these conditions, the stored metadata include the CVE identifier, associated CWE identifier, repository owner and name, fix commit SHA, parent (vulnerable) SHA, total change count, and commit timestamp; the resulting commit set forms the basis for subsequent diff analysis and bug-fix location extraction.

C.2.2 Diff Processing and Bug-Fix Extraction

Each retained fix commit is reduced to a single, source-localized bug–fix region representing the root-cause code modified by the revision. The analysis explicitly models small-patch repairs: commits are constrained to at most eight total modified lines, a threshold that tolerates incidental comment or whitespace edits while ensuring that the retained change captures a compact and causally coherent fragment of the original fix.

Diff processing. Target files are identified as those marked `modified` in the commit and whose extensions fall within the C/C++ source (set defined in Section C.2.1); commits without such files are discarded. For each candidate file, the unified diff between the vulnerable parent and the fix revision is examined hunk by hunk. A hunk is considered code-relevant only if it modifies at least one non-comment, non-whitespace line: blank lines and both single-line (`//`) and block (`/* */`) comments are ignored, and multi-line block comments are tracked to prevent misclassification. Hunks that affect only formatting or comments are removed. After filtering, the commit is retained only if exactly one logical modification region remains; multiple qualifying hunks result in exclusion to preserve a one-to-one mapping between vulnerability instance and defect location.

Bug–fix line-span extraction. For the retained hunk, line spans are derived from the set of changed line numbers. In the vulnerable version, the span covers deleted or modified lines in the parent revision; in the fixed version, it covers the corresponding added or modified lines. Pure additions or deletions are anchored to the nearest changed line in the respective version. Each qualifying hunk yields two aligned records each defined by a canonical location (`repository`, `file path`, `start line`, `end line`) for both vulnerable and fixed file versions. Rather than materializing full repository snapshots, only the affected file is retrieved at the parent and fix commits, preserving the exact code context for downstream SAST execution and snippet extraction.

C.2.3 SAST-Based Training Data Labeling

Training data labeling is performed by executing SAST tools on the source files extracted during patch-based dataset construction (Section C.2.2) and aligning tool warnings with the bug-fix locations. The execution environment, analyzers, and invocation commands are identical to those used during evaluation (Section C.1.3); the primary extension lies in the input specification. Scans are scoped to this dataset-defined file set using the same orchestration logic as in evaluation, differing only in invocation arguments, so that analyzers process exactly the files implicated by vulnerability fixes. SARIF artifacts are normalized using the same pipeline described in Section C.1.1.

cve_id	repo_path	version	file	start	end	tool	rule	label
CVE-2022-1341	vgropp/bwm-ng	b81da650	src/options.c	338	338	Flawfinder	FF1019	correct
		9774f23b		342	342			incorrect

Flagged code (both instances):

```
snprintf(str, strlen(pwd_entry->pw_dir)+14, "%s/.bwm-ng.conf", pwd_entry->pw_dir);
```

The vulnerable version lacks a preceding check on `str`, whereas the fixed version introduces a guard condition. Although the flagged line is identical, its semantic correctness differs across versions, yielding opposing labels.

Table C.1: Example training instances derived from the same CVE and SAST rule. The flagged code is syntactically identical, but contextual differences between vulnerable and fixed versions determine the assigned label.

Normalized SARIF results are then ingested into a tabular representation, where each warning is materialized as a structured row containing its primary location (file path, start line, end line), tool and rule identifiers, message text, and an extracted code snippet. Labeling is performed strictly via *location alignment* between these SAST warnings and ground-truth bug-site spans; tool-reported CWE identifiers or rule semantics are not used for labeling. Within each repository–commit group, SARIF warnings are matched against the corresponding ground-truth file entries using line-range containment. Warnings whose reported location falls entirely within a bug-site span are retained and labeled according to scan context: warnings produced when analyzing the *vulnerable* version are labeled `correct`, while warnings produced when analyzing the *fixed* version are labeled `incorrect`. Warnings outside the dataset-defined file set or failing to overlap any ground-truth span are ignored, and duplicate warnings referring to the same *tool*, *file*, and *start line* are removed. Each retained warning–location pair is materialized as a single training instance. Table C.1 illustrates two such instances derived from the same vulnerability and SAST rule, where the flagged line of code is syntactically identical but differs in semantic correctness due to contextual changes introduced by the fix.

C.3 Reasoning Layer Implementation and Explanation Generation

This section specifies the concrete implementation of the reasoning layer introduced in Section 4.4.3. Section C.3.1 details localized context assembly and model invocation, Section C.3.2

defines the prompt template and instruction variants, and Section C.3.3 describes response processing and SARIF enrichment.

C.3.1 Context Assembly and Model Invocation

Explanation generation operates over the structured warning instances produced during SAST-based training data labeling (Section C.2.3). At this stage, each warning is a normalized record containing its primary location, analyzer metadata, and the corresponding file revision. The reasoning layer performs no additional repository discovery, diff analysis, or warning matching; it consumes these pre-aligned instances directly and transforms each into a bounded model invocation that exposes only localized code and the tool message.

Context assembly. For each warning instance, a localized code view is constructed around the tool-reported line range. The highlighted region corresponds exactly to the warning's start and end lines in the stored file revision. Surrounding context is extracted from a fixed window to provide minimal local context without expanding beyond the immediate scope of the defect: up to ten lines preceding the target start line are included as pre-context, and up to five lines following the target end line as post-context. When explanations are generated for fixed revisions, an explicit offset may be applied to compensate for line shifts introduced by patch edits so that the highlighted region corresponds semantically to the same logical code fragment across revisions. In parallel, the tool message undergoes lightweight normalization before prompt construction: tool-specific identifiers (e.g., CWE tags) are removed, retaining only the free-text description of the suspected issue. The resulting prompt context, therefore, consists solely of (i) localized source code and (ii) a minimally processed description, without cross-file context, repository-level metadata, or ground-truth labels.

Model input and invocation. The normalized warning message and assembled code context are combined into a structured prompt that serves as a template carrying the localized evidence described above. A separate high-level instruction governs the expected response behavior, and depending on the interface, it can be supplied independently of the prompt body. The reasoning layer supports two execution backends through a thin abstraction interface. When no external API token is configured, explanations are generated using a locally hosted `google/gemma-3-1b-it` language model; in this configuration, the instruction and prompt are concatenated into a single input string of the form `"{instruction}\n{prompt}\nAnswer:\n"` and generated deterministically (`do_sample=False`) with a maximum of 128 newly generated tokens to ensure reproducibility. When an API token is available, explanations are generated using OpenAI's `GPT-4.1` model; the instruction is supplied via the API's dedicated instruction field, and the prompt content is provided as the input parameter, with default generation settings. In both configurations, the model is exposed only to the localized code context and the normalized tool message assembled above; no ground-truth labels or benchmark metadata are revealed during explanation generation.

Ablations. Model capability for explanation generation was evaluated, as early results showed limited separation between *correct* and *incorrect* warnings in t-SNE projections and classifier performance. Manual inspection of generated explanations revealed reasoning inconsistencies. In `CVE-2022-1341`, the buggy explanation correctly identified a visible guard (`pwd_entry != NULL`) but failed to surface the key differentiator—the absence of a `str != NULL` check. Conversely, the safe explanation sometimes contradicted the provided context by claiming no guard was shown despite the presence of an explicit `if(!str) ... exit` condition. Such internal contradictions and vague phrasing (e.g., generic "Operation: call" statements or inconsistent operand naming) introduced noise that reduced stability and impaired downstream learning.

Several models were evaluated for explanation generation: `GPT-4.1-nano`, `GPT-4.1-mini`, `GPT-4.1`, `GPT-5-nano`, and `GPT-5-mini`. The `GPT-4.1` variants support a 1M-token context window, while the `GPT-5` nano/mini variants support a 400,000-token context window; thus, input length was not a limiting factor in either case. Only `GPT-4.1` consistently resolved guard-level distinctions and avoided the contradictions observed in smaller variants. The `GPT-5` nano/mini variants exhibited higher latency (approximately 30 seconds per request) and increased cost with gains in reasoning fidelity. Since `GPT-4.1` provided the consistent explanations with acceptable latency and cost, and it was selected for the rest of the experiments. Across the entire evaluation stage, a total of **2,417,050 tokens** were processed over **3,006 requests**, at a cost of **$4.29**.

C.3.2 Prompt Template and Instruction Variants

Explanation generation uses a fixed prompt template and a small set of instruction variants that guide how the model responds during training and deployment. This subsection describes the structure of the prompt and explains how different instruction settings control the form and content of the generated explanations.

Prompt template. Each explanation request follows a fixed textual template that separates tool guidance from code context and explicitly marks the statement reported by the analyzer. The normalized warning message appears first and serves as a focus hint. It is followed by a C/C++ snippet that embeds the localized code context. Within this snippet, the lines preceding the analyzer-reported range are included verbatim as pre-context, the highlighted statement corresponding exactly to the reported start and end lines is inserted and explicitly marked, and the subsequent lines are appended as post-context. The inline highlight marker anchors the model's attention to the analyzer-reported location while preserving surrounding structural cues. Listing C.5 shows the implicit prompt template used to assemble these components. The template remains invariant across all phases of the pipeline; only the surrounding instruction, supplied separately during model invocation, varies depending on the generation context.

Listing C.5: Prompt Template for SAST Warning Explanation

```
Tool message (focus hint only, do not quote):
<normalized_warning_message>

C/C++ snippet (the highlighted line is marked):

<context_before>
__HIGHLIGHT__ <target_statement>
<context_after>
```

Instruction variants. All explanation requests are paired with a high-level instruction that constrains both explanation form and evidence patterns. While the prompt template remains invariant, the instruction text varies depending on whether explanations are generated for supervised training or deployment triage. All variants share a common structural header and output specification, shown in Listing C.6. In addition to prescribing the single-sentence format, the body defines the semantic slots (`FORM_WORD`, `OPERATION_PHRASE`, `FOCUS_OPERAND_PHRASE`, and `EVIDENCE_PHRASE`) and prohibits speculation, vulnerability terminology, or quoting the tool message. The concrete instruction variants differ in the admissible `FORM_WORD` values and in the allowed patterns for `EVIDENCE_PHRASE`, as follows:

- **Correct-warning instruction (training, vulnerable revision)[6].** Used when a tool warning overlaps a bug-site location in the vulnerable version. The instruction restricts `FORM_WORD` to either `Missing` or `Incorrect`. Correspondingly, the admissible `EVIDENCE_PHRASE` patterns are limited to those expressing either absence of constraint or an insufficient constraint attempt;
- **Incorrect-warning instruction (training, fixed revision)[7].** Used when a tool warning overlaps the corresponding location in the fixed version. The instruction restricts `FORM_WORD` to `Proper`. The admissible `EVIDENCE_PHRASE` patterns are limited to expressions of explicit or structurally sound constraint;
- **Deployment instruction (evaluation/triage)[8].** Used during evaluation and deployment, where ground truth about defect presence is unavailable. The instruction permits all three `FORM_WORD` values (`Proper`, `Incorrect`, or `Missing`) and includes the full set of admissible `EVIDENCE_PHRASE` patterns spanning both constraint-establishing and constraint-failing cases. It additionally clarifies the distinction between `Missing` and `Incorrect`.

[6] https://github.com/epicosy/phd_thesis/blob/main/config/chapter_4/tp_instruction.txt

[7] https://github.com/epicosy/phd_thesis/blob/main/config/chapter_4/fp_instruction.txt

[8] https://github.com/epicosy/phd_thesis/blob/main/config/chapter_4/instruction.txt

Listing C.6: Shared Instruction Header and Output Format

```
You are given a contiguous C/C++ snippet where one line is marked
as the highlighted line, plus a tool-provided message.

The tool message indicates which operation/operand aspect is of
interest (the focus).
Do not quote or paraphrase the message.
Do not use vulnerability or weakness terminology.

Write exactly one sentence using this structure:

<FORM_WORD> <OPERATION_PHRASE> over <FOCUS_OPERAND_PHRASE, where
the code <EVIDENCE_PHRASE>.

Choose FORM_WORD: ...
OPERATION_PHRASE: Name the operation of the ...
FOCUS_OPERAND_PHRASE: Name the focused operand ...
EVIDENCE_PHRASE must be one of: ...

Do not speculate about impact or intent.
```

Ablations. Instruction design was iteratively refined to improve separation between *correct* and *incorrect* warnings, evaluated through qualitative inspection of explanations and t-SNE projections of their embeddings. The initial instruction emphasized step-by-step behavioral narration and discussion of warning trigger conditions and security-relevant properties. The assumption was that richer narratives would expose semantic differences. In practice, explanations were verbose and pattern-oriented, with strong overlap in t-SNE space and near-chance classifier performance.

Refinement proceeded in stages. **First**, explanations were normalized around observable behavior and restructured to encode *assumptions* vs. *enforced constraints*. **Second**, a four-field schema (operation, operands, guards, assumptions) reduced ambiguity but increased verbosity and variance. **Third**, the schema was tightened into constrained behavioral forms (*Proper / Incorrect / Missing*) with an explicit focus selector over operation–operand pairs, and then collapsed into a single fixed-clause sentence to reduce lexical variability. Focus-selection rules were further refined to prioritize checkable operands (e.g., destination buffers, size arguments) and introduce fallback logic for generic tool messages, improving alignment between explanations and labels.

Finally, **label-conditioned generation** was introduced as a controlled ablation. Two modes were defined: **Strict**, which applied a unified deployment instruction to all samples, and **Idealized**, which used label-aware prompts during training to explicitly surface either missing constraints (true positives) or enforced constraints (false positives), while preserving the same output structure. Training on *Idealized* explanations produced clearer clustering and higher classification performance than *Strict* alone, indicating that **structure**, **constraints**, and **focus**—rather than

model capacity—were the primary factors limiting separability under limited data.

C.3.3 Response Processing and SARIF Enrichment

After model invocation (Section C.3.1), each generated explanation is associated with its corresponding warning instance in the working dataset via the stable `location_id`. Explanations are produced per warning row and stored alongside the structured warning record, preserving the linkage between the tabular representation and the original SARIF location. The reasoning layer thus yields an augmented warning dataset that can be reintegrated into SARIF artifacts while maintaining traceability to the original analyzer output.

Response processing. Response handling differs slightly between the local and remote backends. For the remote backend (`gpt-4.1`), the API response is used directly as the explanation text. For the local backend (`google/gemma-3-1b-it`), minimal structural post-processing is required due to the concatenated prompt format. After decoding and removing special tokens, the output is split at the delimiter `"Answer:\n"`. If the delimiter occurs exactly once, the subsequent segment is extracted as the explanation, Markdown code fences (e.g., ` ``` `) are removed, and leading or trailing whitespace is trimmed. Outputs that do not match the expected structure are discarded. No semantic rewriting or normalization is performed beyond this extraction.

SARIF enrichment. Generated explanations are reintegrated into the original SARIF artifact using `location_id` as the join key. The enrichment procedure iterates over SARIF runs and results, renames the tool driver to denote the AI-augmented variant, and replaces result messages for locations with available explanations. Locations without a corresponding explanation are omitted from the enriched result set. The modified SARIF object is serialized to JSON and written to a new artifact with the suffix `.ai.sarif`. This process produces AI-augmented SARIF files in which analyzer-generated diagnostics are replaced by structured behavioral explanations, while preserving file paths, line ranges, and schema compliance. The enriched artifacts remain compatible with standard SARIF tooling. Within the broader pipeline, the generated explanations additionally serve as the sole textual input to the actionable-warning classifier described in Section 4.4.3, ensuring that downstream triage operates over a uniform behavioral representation rather than tool-specific rule semantics.

C.4 Supervised Triage Model and SARIF-Level Filtering

This section specifies the implementation of the supervised triage stage introduced in Section 4.4.3. Section C.4.1 details CVE-level stratified dataset preparation, Section C.4.2 describes transformer fine-tuning and evaluation, and Section C.4.3 outlines SARIF-level filtering for triaged report generation.

C.4.1 Training Dataset Preparation

The triage classifier is trained on the AI-augmented warning dataset produced after explanation generation (Section C.3.3), which extends the labeled SAST warning corpus described in Section C.2.3. At this stage, each row corresponds to a single warning instance aligned to a bug-site span and enriched with a behavioral explanation. The triage problem is formulated as binary sentence classification over the `explanation` field, with labels mapped from the original scheme (`correct`, `incorrect`) to numeric identifiers (1 and 0, respectively). The dataset is consumed as produced by the preceding stages; no additional feature engineering, rule-based filtering, or structural transformations are introduced.

Training and validation splits are constructed using stratified sampling at the CVE level. Rather than splitting warnings independently, all warnings sharing the same `CVE-ID` are kept together, ensuring that buggy and fixed instances of a given vulnerability are never separated across splits. To control the composition of CVEs across partitions, each CVE is summarized by the presence of `correct` and `incorrect` warnings and assigned to one of three strata: CVEs containing only `correct` warnings, only `incorrect` warnings, or a mixture of both. Operationally, the stratum is computed as $(\mathbb{1}[n_{\text{correct}} > 0] + \mathbb{1}[n_{\text{incorrect}} > 0])$, yielding values in $\{1,2\}$. A stratified shuffle split (10% validation, fixed random seed) is then applied over these CVE-level strata, and the resulting CVE assignments induce row membership in each partition. A final overlap check enforces that no CVE appears in both splits, preventing information leakage between dependent bug–fix pairs.

C.4.2 Model Architecture, Configuration, and Training

The triage is implemented as a transformer-based sentence classifier fine-tuned on the generated explanations. The pretrained encoder `sentence-transformers/all-MiniLM-L6-v2`[9] maps sentences and short paragraphs into a 384-dimensional dense vector space and was originally trained using a contrastive objective over large-scale sentence-pair data. Its MiniLM-L6-H384 architecture balances representational capacity and computational efficiency, making it well-suited for short-text classification tasks such as explanation-based triage. For this task, the pretrained encoder is extended with a sequence-classification head configured for binary prediction.

Input preparation operates over the `explanation` field. The dataset is converted into a HuggingFace `Dataset` object containing explanation text and numeric label identifiers, tokenized using the model's pretrained tokenizer with padding to a fixed maximum length and truncation enabled (consistent with the 256 word-piece limit), and formatted for PyTorch. Supervised fine-tuning is performed using `AutoModelForSequenceClassification` with two output labels and a single-label objective. Optimization uses AdamW with a learning rate of 2×10^{-5}, weight decay of 0.01, and training over 3 epochs, with mini-batch sizes of 32 (training) and 64 (evaluation). Evaluation and checkpointing occur at the end of each epoch, and the model achieving the highest macro-averaged F1 score on the validation set is restored at completion. Predictions are

[9] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

obtained via arg max over output logits, and performance is reported using accuracy and macro-averaged F1, the latter ensuring balanced assessment across `correct` and `incorrect` classes.

Ablations. Fine-tuning was evaluated under *strict-only*, *ideal-only*, and *mixed* explanation settings with varying epoch counts. Under the *strict-only* setting, validation performance remained limited (approximately 0.57 accuracy and 0.53 macro-F1 at **3 epochs**), with no false-positive reduction. Training on *ideal-only* explanations for **3 epochs** achieved **0.71** accuracy and **0.68** macro-F1, with stable filtering behavior (removal of **40** false positives while retaining **3** true positives), while extending to **5 epochs** reduced deployment stability despite lower training loss. In contrast, *mixed* training yielded lower performance at **3 epochs** (0.53 accuracy, 0.40 macro-F1) and unstable filtering; longer training up to **10 epochs** improved validation metrics (peaking at 0.75 accuracy and 0.73 macro-F1 at epoch 5) but did not consistently improve deployment behavior. Overall, **3 epochs** provided the most consistent validation performance, and explanation distribution differences had a stronger effect than additional optimization steps.

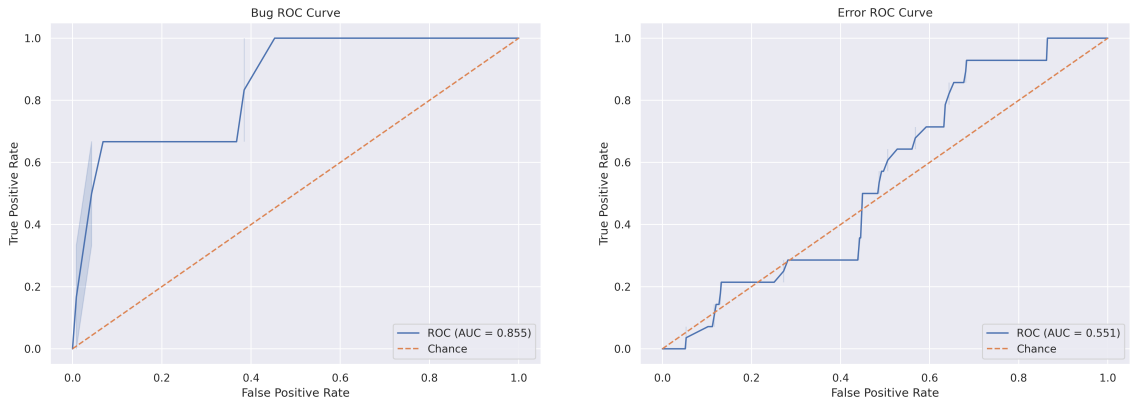
C.4.3 Classifier-Based SARIF Filtering

After training, the classifier is integrated as a post-processing step over AI-augmented SARIF files produced by the reasoning layer (Section C.3.3). For each repository snapshot and tool, the corresponding `.ai.sarif` file is processed independently. Static analysis is not re-executed; the triage stage operates exclusively over existing SARIF results enriched with behavioral explanations. The output is a new SARIF artifact written alongside the original scan, using a distinct tool identifier (e.g., `<tool>_triaged`) to distinguish the filtered variant from both the raw analyzer output and the explanation-augmented artifact.

Filtering is performed at the level of SARIF `results`. For each run, the result list is passed to a `FalsePositiveFilter`, which extracts the explanation text from the primary location message of each result. Explanations are processed in batch using the stored tokenizer with fixed-length padding and truncation, and passed through the fine-tuned sequence-classification model. Predictions are obtained via arg max over the output logits (unless an explicit probability threshold is specified). Results predicted as `correct` are retained, while those predicted as `incorrect` are removed; if no explanations are present for a run, the result set is treated as empty. The SARIF structure is then rewritten by replacing `run.results` with the filtered list and updating `tool.driver.name` to reflect the triaged variant. To preserve structural consistency, `tool.driver.rules` is pruned so that only definitions corresponding to retained results. The modified SARIF object is serialized to a new file with the suffix `.triaged.sarif`. File paths, line ranges, rule identifiers, and other location metadata remain unchanged throughout; the classifier acts only as a decision layer that removes predicted non-actionable warnings.

C.4.4 ROC Analysis and Decision Threshold Characterization

To characterize the classifier's decision behavior, we compute Receiver Operating Characteristic (ROC) curves and the corresponding Area Under the Curve (AUC) for both bug-site and error-site settings. Using the same matched SARIF artifacts described in Section C.4.3, we extract explanation texts and assign binary oracle labels based on exact-line containment within the annotated ground-truth spans. The fine-tuned model is then queried in score-returning mode to obtain posterior probabilities for the `correct` class, and ROC curves are constructed by sweeping the decision threshold over $[0, 1]$. This analysis evaluates the classifier as a ranking function—i.e., its ability to separate actionable from non-actionable warnings independently of a fixed operating point—and provides diagnostic context for the precision–recall trade-offs observed in Section 4.5.4, particularly the recall degradation at error sites.



(a) Bug-site ROC (AUC = 0.855)

(b) Error-site ROC (AUC = 0.551)

Figure C.1: ROC curves for bug-site and error-site triage. The classifier exhibits strong separability at defect locations (AUC = 0.855) but near-chance performance at error locations (AUC = 0.551).

ROC comparison across oracle levels. As shown in Figure C.1(a), the classifier achieves strong ranking performance at bug sites (AUC = 0.855), with the ROC curve rising steeply at low false-positive rates, indicating clear score separation between actionable and non-actionable warnings. In contrast, Figure C.1(b) shows that performance at error sites is only marginally above chance (AUC = 0.551), with the curve remaining close to the diagonal and reflecting substantial overlap in score distributions. This contrast mirrors the deployment results in Section 4.5.4: triage preserves all true positives at defect locations due to high separability, whereas at manifestation locations weaker score discrimination leads to recall degradation when false positives are filtered.

C.5 Evaluation Protocol and Matching Details

This section specifies the procedures underlying the evaluation design described in Section 4.4.4. Section C.5.1 details oracle matching, rule normalization, and deduplication, while Section C.5.2

documents the mapping from analyzer rules to CWE identifiers and their abstraction to Bugs Framework operations for assessment.

C.5.1 Exact-Line Ground-Truth Matching and Warning Deduplication

Evaluation is performed by matching tool-reported SARIF locations against normalized benchmark locations and aggregating the resulting matches and misses for metric computation. The procedure operates at the repository snapshot level, defined by `(owner, repo, commit_sha)`, and consumes three inputs: the normalized datasets with ground-truth locations (Section C.1.1), the corresponding SARIF scan outputs (Section C.1.3), and checked-out file snapshots for path validation. For each snapshot, SARIF results are filtered, grouped by file, compared against benchmark targets using an exact-line oracle criterion, and deduplicated prior to metric aggregation.

SARIF selection and file-scoped processing. Evaluation iterates SARIF files over repository snapshots grouped by `(owner, repo, commit_sha)`. For each snapshot, SARIF artifacts are loaded from the scan directory, excluding explanation-enriched variants (e.g., files containing `.ai`) to ensure that only raw analyzer outputs are evaluated. Results are processed per SARIF file and grouped by repository-relative file path, after validating that each reported location resolves to an existing file in the checked-out snapshot. Evaluation is restricted to benchmark-referenced files, i.e., those appearing in the ground-truth dataset for the given snapshot; SARIF results for other files are ignored. Path-level ignore lists (e.g., for test or third-party directories) are applied during input construction but do not alter the oracle matching logic described below.

Oracle matching, rule normalization, and deduplication. For each evaluated file, ground truth is derived from the normalized benchmark dataset (Section C.1.1) and represented as inclusive line spans `[start_line, end_line]`, grouped by ground-truth CWE identifier `cwe_id`. Each SARIF result location yields a target region defined by its reported start and end lines; locations lacking complete line information are skipped. Matching is performed using a containment check with configurable margins:

$$\texttt{target.start} \geq (\texttt{gt.start} - \texttt{MARGIN_BEFORE_START_LINE}) \wedge \texttt{target.end} \leq (\texttt{gt.end} + \texttt{MARGIN_AFTER_END_LINE})$$

In the reported experiments, `MARGIN_BEFORE_START_LINE` and `MARGIN_AFTER_END_LINE` are set to `0`, yielding strict containment; in the common single-line case, this reduces to an exact-line match. The margin parameters can be adjusted to permit relaxed matching if desired. Tool rule identifiers are normalized prior to evaluation, and a tool-specific mapping associates normalized rule identifiers with CWE identifiers; this association is recorded for analysis but does not gate location-based matching. Matched and unmatched locations are accumulated as *matches* and *misses*, respectively. During result materialization, an initial deduplication removes identical diagnostics using {`tool`, `file`, `rule_id`, `start_line`, `end_line`}. Prior to metric computation,

both matches and misses are deduplicated again using {`tool`, `file`, `start_line`, `end_line`}, ensuring that multiple diagnostics referring to the same region are counted once per tool. The total number of ground-truth instances is computed as the number of unique `(owner, repo, commit_sha, file, start_line, end_line)` tuples in the evaluation dataset and is used to derive false negatives as $\mathrm{FN} = |\mathrm{GT}| - \mathrm{TP}$.

C.5.2 Mapping Tool Rules to Bugs Framework Operations

Table C.2 summarizes the rule-to-mechanism mappings used for assessment. The table is intentionally non-exhaustive: it lists only analyzer rules that produced at least one location-matched warning under the benchmark oracle(s), focusing on mechanisms directly relevant to the reported results. For each rule, the associated CWE identifier is either taken from the tool's native mapping or assigned based on the closest semantic correspondence between the rule description and the CWE definition. The CWE therefore serves as the linking layer between tool diagnostics and benchmark semantics. BF operations are then derived from the operation-level behavior implied by the associated CWE and, where necessary, the rule description. CWE rationales document how the rule–CWE relationship was established (e.g., tool-provided, closest semantic match, direct description alignment), while BF rationales justify the abstraction from weakness taxonomy to execution-level mechanism.

Tool	Rule	CWE		Bugs Framework	
		ID	Rationale	Op.	Rationale
Cppcheck	`cstyleCast`	704	Closest semantic match to rule intent (type cast)	Cast	Derived from associated CWE semantics
Cppcheck	`shadowVariable`	563	Closest semantic match to outer state shadowing	Declare	Shadowing originates at variable declaration
Flawfinder	`FF1004`	120	Tool-provided CWE mapping	Write	Derived from associated CWE semantics
Flawfinder	`FF1019`	134	Tool-provided CWE mapping	Validate	Derived from associated CWE semantics
CodeQL	`declaration-hides-variable`	1126	Closest semantic match to variable hiding	Declare	Rule and CWE focus on declaration semantics
CodeQL	`cpp/mistyped-function-ar...`	686	Rule name/description directly indicates CWE	Call	Derived from associated CWE semantics
Semgrep	`interesting-api-calls`	676	Risky API usage consistent with CWE	Call	Function call/usage is semantically improper
Semgrep	`mismatched-memory-man...`	762	Rule description matches CWE definition	Deallocat	Derived from associated CWE semantics
Semgrep	`unsafe-ret-snprintf-vs...`	252	Rule desc. indicates misused return value	Verify	Check hints at semantic correctness of the value

Table C.2: Rule-to-mechanism mappings for analyzer rules that produced location-matched warnings. CWE rationales reflect tool-level weakness interpretation; BF rationales justify abstraction to operation-level semantics.

Rules frequently do not correspond to a single, precisely delimited CWE entry, and many analyzer diagnostics conflate multiple related weakness patterns. Each rule was therefore mapped to the closest semantically aligned CWE, and the resulting assignments should be interpreted as approximate taxonomic alignments rather than exact one-to-one weakness equivalences. The subsequent mapping from CWE to BF operation likewise requires abstraction across representation levels and is not strictly mechanical. In particular, the use of *Call* departs slightly from the orthodox BF definition, where *Call* denotes correct name resolution and binding of a function. Here, *Call* is used more broadly to mark the invocation site as the locus of the weakness, even when binding is technically correct but the selected implementation is semantically inappropriate in context. While this extends the formal definition, *Call* remains the closest available BF operation to capture contextual API misuse without introducing additional operations.