



UNIVERSIDADE D
COIMBRA

José Alexandre D'Abruzzo Pereira

**SOFTWARE SECURITY CHARACTERIZATION
THROUGH STATIC DATA ANALYSIS**

PhD Thesis in Informatics Engineering, Architectures, Networks and
Cybersecurity, advised by Professor Marco Paulo Amorim Vieira, and presented to
the Department of Informatics Engineering of the Faculty of Sciences and
Technology of the University of Coimbra

December, 2023



DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

José Alexandre D'Abruzzo Pereira

**SOFTWARE SECURITY CHARACTERIZATION
THROUGH STATIC DATA ANALYSIS**

**PhD Thesis submitted to the University of Coimbra
Advised by Professor Marco Vieira**

December, 2023



DEPARTAMENTO DE
ENGENHARIA INFORMÁTICA
FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE
COIMBRA

José Alexandre D'Abruzzo Pereira

**CARACTERIZAÇÃO DE SEGURANÇA DE
SOFTWARE POR ANÁLISE DE DADOS
ESTÁTICOS**

**Tese de Doutoramento submetida à Universidade de Coimbra
Orientada pelo Professor Marco Vieira**

Dezembro, 2023

Projects and Funding

The work presented in this thesis was carried out within the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC) in the context of the following projects:

- **ATMOSPHERE:** Adaptive, Trustworthy, Manageable, Orchestrated, Secure Privacy-assuring Hybrid, Ecosystem for REsilient Cloud Computing; a 24-month project aiming at the design and development of an ecosystem of a framework, platform and application of next generation trustworthy cloud services on top of an intercontinental hybrid and federated resource pool. The framework considers a broad spectrum of properties and their measures. The platform supports the building, deployment, measuring and evolution of trustworthy cloud resources, data network and data services. ATMOSPHERE is funded by the European Union under the Cooperation Programme, Horizon 2020 grant agreement no. 777154.
- **METRICS:** Monitoring and Measuring the Trustworthiness of Critical Cloud Systems; cloud is pervasive nowadays, but its adoption in critical systems is limited by trust issues, mainly influenced by security, dependability, privacy, fairness and transparency concerns, as per the recent GDPR regulation. As an evolving concept, the trustworthiness of the system must be continuously monitored and measured, but there is a lack of means to do that in cloud environment. This project aims to propose a framework and means for monitoring and assessing the trustworthiness of cloud systems. This includes the definition of trustworthiness properties, their continuous measurement and analysis. METRICS is co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the Fundo Europeu de Desenvolvimento Regional (FEDER) through Portugal 2020 - Programa Operacional Competitividade e Internacionalização (POCI-01-0145-FEDER-032504).
- **AIDA:** Adaptive, Intelligent and Distributed Assurance Platform; the project aims at improving a platform used by Mobileum for integral risk management in companies. This platform ensures revenue, corporate conditions and fraud control for companies. Thanks to the newest version of the platform, developed by AIDA, companies will be able to collect and monitor data in an extremely flexible way, with real-time guarantees, security and reliability. AIDA is co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalization – COMPETE 2020 (POCI-01-0247-FEDER-045907) and by the Portuguese Foundation for Science and Technology under CMU Portugal Program.

- **TalkConnect:** The *Voice Architecture over Distributed Network* project aims at researching and implementing a new solution which combines multiple direct telco connection with third party Communication Platform as a Service (CPaaS) providers with global coverage. One of the main corner-stones of this work is related to ensuring the security of the platform and of the communications. More precisely, these are cloud environments that typically use microservice architectures and virtualisation environments to deploy such application exacerbating security concerns that need to be addressed. TalkConnect is funded by the FCT – agreement no. POCI-01-0247-FEDER-039676.
- **ADVANCE - Addressing Verification and Validation Challenges in Future Cyber-Physical Systems:** The scientific objective of the ADVANCE project is to conceive new approaches to support the Verification and Validation (V&V) of Cyber-Physical Systems (CPS). It will explore techniques, methods, and tools applicable to different phases of the system lifecycle, but always with the final objective of improving the effectiveness and efficacy of the V&V process. ADVANCE is funded by European Union’s Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement no. 823788.

This work has been partially supported by the following grants:

- **Ph.D. grant:** Foundation for Science and Technology (FCT) Grant number: 2020.04503.BD.
- **Research grant: ATMOSPHERE - H2020 - EUB-01-2017 - 777154:** Grant number: DPA-18-258 / Period: 10/04/2018 - 09/01/2019;
- **Research grant: METRICS - POCI-01-0145-FEDER-032504:** Grant number: DPA-18-884 / Period: 15/01/2019 - 14/07/2019.



Cofinanciado por:



Acknowledgements

This Ph.D. work was only possible due to the many people who supported me throughout the years of this journey.

I want to thank my advisor and friend, Professor Marco Vieira. Thank you for the invitation to pursue this Ph.D., which made me grow personally and professionally. It was very challenging due to the unexpected things that happened (again, both personally and professionally), but it was a very rewarding process. The numerous meetings that we had always inspired me. Your calm and wisdom made me continue working toward my goal. Also, thank you for letting me become part of the Software and Systems Engineering (SSE) group at the Centre for Informatics and Systems of the University of Coimbra (CISUC). This allowed me to meet great people in the group and to participate in several research projects with amazing researchers from other institutions in Portugal and abroad.

Throughout my life, my parents have always supported my decisions regarding my education and for accepting me the way I am. Without their daily support (even far from Coimbra), this journey would not have been possible. Thank you for the endless support and for hearing me every time I complained about something about the Ph.D. or about my (second) adaptation in Portugal. Thank you for being comprehensive whenever I could not be in Campinas during this period.

Since the moment that I decided to go for the Ph.D., one person has always been by my side: Julio. I could not imagine this path without your endless support to continue working toward my goal. Every time I complained about the Ph.D., you always advised me: “The Ph.D. is like this; you are doing your best”. Thank you so much for listening to me, helping me, and taking care of me, especially when I got sick. My life with you is much better, and you make me a better person.

I also had support from my colleagues (now friends) from the laboratory and from the research center: Charles, João Campos, Nádia, Gonçalo, Ivanilson, José Flora, Jessica, and Iury. Thank you for the several discussions that we had throughout those years and for sharing the anguish that we faced during those years. Rui and Paulo are two friends whose friendship started in the laboratory but continues present in my life even without the daily involvement. Felipe Duarte is also a friend from the laboratory that is still present in my life.

My friends were also essential during this time. All my friends in Brazil were by my side, even with the distance. Evandro, Mari, Fábio, Suzana, Gustavo, Laís, Tuany, and Camila, just to name a few of them. I also want to thank my friends from Portugal whom I met during the masters, especially Sofia, Teresa, David, and Sandra. There were people who became my friends in Coimbra outside the University: Emerson, Thiago, Elias, and Sergio. Last but not least, the friends that joined my life this year and made my life much better: Dener, Pedro, and Hugo.

There is someone who saw the beginning of my Ph.D. but could not see the end: my grandmother and godmother. I could not be there in Brazil when you departed: it was when the COVID-19 pandemic started, and I was sick here in Portugal. I know that you would be very proud of me at this point in my life.

Abstract

Modern enterprises rely on software systems to run their business: financial, healthcare, government, and e-commerce, among many others. However, many systems are deployed with vulnerabilities caused by a design flaw or an implementation bug. The malicious exploitation of those security vulnerabilities may lead to various problems with financial or legal implications.

Vulnerability detection techniques can be divided into two main groups: *static techniques* and *dynamic techniques*. The most known static technique is Static Code Analysis (SCA) that reports potential problems (alerts) without requiring the execution of the code. There are other more recent techniques, for example, based on ML, where static properties extracted from the source code are used as features to predict vulnerabilities. On the dynamic techniques, the most known is Software Penetration Testing (SPT) that simulates attacks in a controlled environment.

Vulnerability detection techniques have strong limitations. Tools are frequently too expensive for most organizations, and they either report many false positives or false negatives. Consequently, developers are required to spend a considerable amount of time analyzing the reported cases without being sure that all vulnerabilities have been detected. **This thesis advances the state-of-the-art on the characterization of software code units from a security vulnerability perspective, making use of static data from the source code.**

The first contribution is a *dataset of static data on vulnerability fixes*. This dataset includes vulnerabilities from five open-source C/C++ projects (Linux Kernel, Mozilla, Xen, Apache httpd, and glibc), and static data (Software Metrics (SMs) and alerts from Static Analysis Tools (SATs)) extracted from the vulnerable and neutral versions of the code. Vulnerabilities are organized into categories, devised based on the improper or lack of use of the OWASP best practices.

To better understand static vulnerability detection, we present two studies. In the first, vulnerabilities from the Mozilla project are used to *study the performance of SATs* in detecting different types of software vulnerabilities. Results confirm that none of the SATs used (CppCheck and Flawfinder) can be trusted, as they fail to detect a large number of vulnerabilities while raising a large number of false positives. Developing on these observations, in the second study, we *analyze the characteristics of a set of buffer overflow vulnerabilities* from the Linux Kernel, Mozilla, and Xen projects, using both static data and a classification of the vulnerabilities with ODC. The main findings are that there is no strong correlation between the static information considered and the occurrence of vulnerabilities and that fixing overflow vulnerabilities typically increases the code size and complexity.

Pursuing the use of ML to detect vulnerabilities, we present two studies. In the first one, *classical ML algorithms are used to predict the presence of vulnerabilities in files* of the Mozilla project. Both SMs and SAT alerts are used as features for the ML algorithms (DT, RF, XGB, and Bagging). In the second, we follow an approach based on *deep learning to detect vulnerabilities*, grounded on DGCNN and VGG networks. In this case, features extracted from the CFG of functions of the

Linux Kernel project are used. We conclude that none of the Machine Learning (ML) approaches considered can predict software vulnerabilities with acceptable performance. In fact, although we can obtain good precision and good recall in some cases, no configuration allowed us to obtain both simultaneously.

The low performance of SATs and ML observed in the studies above led us in a different direction. The *Security Characterization of Open-source functions using Logic Scoring of Preference (SCOLP)* allows categorizing code units using static information, including SMs and memory management-related attributes extracted from the CFG. In practice, SCOLP assigns the code units into priority groups, from most critical to least critical, based on the output of a set of Quality Models (QMs) focusing on different properties. We demonstrate the approach with functions of the Linux Kernel project and rely on the judgment of security experts to validate the outputs. Results show that SCOLP provides categorizations similar to security experts.

The last contribution is the *Trustworthiness Monitoring & Assessment (TMA)* framework, which brings self-adaptation abilities to software systems. TMA relies on QMs to characterize properties of interest regarding the managed system. The integration with the managed system happens through probes and actuators. The framework can be used at design-time (*e.g.*, to collect security evidence) and at run-time (to detect and mitigate potential issues; *e.g.*, intrusions). To demonstrate TMA, we present a scenario of scaling containers in a microservice application.

Keywords

Software Security, Software Vulnerabilities, Software Metrics, Static Code Analysis, Machine Learning

Resumo

As empresas modernas dependem de sistemas de software para gerir os seus negócios nas mais diversas áreas. No entanto, muitos sistemas são implantados com vulnerabilidades causadas por uma falha de design ou um defeito de implementação. A exploração maliciosa dessas vulnerabilidades de segurança pode levar a vários problemas com implicações financeiras ou jurídicas.

As técnicas de deteção de vulnerabilidades podem ser divididas em dois grupos: *técnicas estáticas* e *técnicas dinâmicas*. A técnica estática mais conhecida é a análise estática de código, que reporta possíveis problemas (alertas) sem requerer a execução do código. Existem outras técnicas mais recentes, por exemplo baseadas em ML, onde propriedades estáticas extraídas do código-fonte são usadas como recursos para prever vulnerabilidades. Nas técnicas dinâmicas, a técnica mais conhecida é baseada em testes de penetração, que simulam ataques em um ambiente controlado.

As técnicas de deteção de vulnerabilidades existentes têm fortes limitações. As ferramentas são frequentemente demasiado dispendiosas para grande parte das organizações e reporta um elevado número de falsos positivos ou de falsos negativos. Consequentemente, os desenvolvedores são obrigados a gastar uma quantidade considerável de tempo a analisar alertas, sem ter certeza de que todas as vulnerabilidades foram detetadas. **Esta tese avança o estado-da-arte na caracterização de unidades de código de uma perspetiva de vulnerabilidades de segurança, fazendo uso de dados estáticos extraídos do código-fonte.**

A primeira contribuição é um *repositório de dados estáticos sobre correções de vulnerabilidades*. Este repositório inclui vulnerabilidades de cinco projetos C/C++ de código aberto (Linux Kernel, Mozilla, Xen, Apache httpd e glibc) e dados estáticos (métricas de software e alertas de ferramentas de análise estática) extraídos das versões vulneráveis e neutras do código. As vulnerabilidades estão organizadas em categorias, definidas com base no uso impróprio ou na falta de uso das melhores práticas da OWASP.

Para melhor compreender a deteção por análise estática, apresentamos dois estudos. No primeiro, as vulnerabilidades do projeto Mozilla são usadas para *analisar o desempenho de analisadores estáticos na deteção de diferentes tipos de vulnerabilidades de software*. Os resultados confirmam que nenhuma das ferramentas usadas (Cp-pCheck e Flawfinder) é confiável, pois falham na deteção de um grande número de vulnerabilidades enquanto reportam um elevado número de falsos positivos. Com base nestas observações, no segundo estudo, *analisamos as características de um conjunto de vulnerabilidades de buffer overflow* dos projetos Linux Kernel, Mozilla e Xen, usando dados estáticos e uma classificação das vulnerabilidades com ODC. As principais descobertas são: *i)* não se observa uma forte correlação entre os dados estáticos consideradas e a ocorrência de vulnerabilidades, e *ii)* a correção de vulnerabilidades de buffer overflow normalmente aumenta o tamanho e a complexidade do código.

Focando na utilização de ML para detetar vulnerabilidades, apresentamos dois estudos. No primeiro, usamos *algoritmos clássicos para prever a presença de vulnerabilidades em ficheiros* do projeto Mozilla. Métricas de software e alertas de análise estática são usados como recurso para os algoritmos de ML (DT, RF, XGB e Bagging). No segundo estudo, seguimos uma abordagem baseada em *deep learning para detetar vulnerabilidades*, assente em redes DGCNN e VGG. Neste caso, são utilizados recursos extraídos do CFG de funções do projeto Linux Kernel. Os estudos levam a concluir que nenhuma das abordagens consideradas é capaz de prever vulnerabilidades de software com desempenho aceitável. De facto, embora possamos obter uma boa precisão e uma boa sensibilidade em alguns casos, nenhuma configuração permitiu obter ambos em simultâneo.

O baixo desempenho das abordagens baseadas em análise estática e em ML observado nos estudos acima, levou o trabalho a seguir uma direção diferente. A *técnica SCOLP* permite categorizar unidades de código usando informações estáticas, incluindo métricas de software e atributos relacionados à gestão de memória extraídos do CFG. Na prática, esta técnica distribui as unidades de código em grupos de prioridade, do mais crítico ao menos crítico, com base no resultado de um conjunto de modelos de qualidade com foco em diferentes propriedades. A abordagem é demonstrada em funções do projeto Linux Kernel, tendo o julgamento de especialistas em segurança sido usado para validar as classificações. Os resultados mostram que o SCOLP fornece categorizações semelhantes às dos especialistas.

A última contribuição é a *abordagem e plataforma TMA*, que traz capacidades de auto-adaptação para sistemas de software. A solução baseia-se em modelos de qualidade para caracterizar propriedades de interesse em relação ao sistema alvo. A integração com esse sistema dá-se através de sondas e atuadores. A plataforma pode ser usada durante o desenho (por exemplo, para recolher evidências de segurança) e em tempo de execução (para detetar e mitigar potenciais problemas, por exemplo, intrusões). Para demonstrar o TMA, apresentamos um cenário de dimensionamento de contentores em uma aplicação baseada em micro-serviços.

Palavras-Chave

Segurança de Software, Vulnerabilidades de Software, Métricas de Software, Análise Estática de Código, Aprendizagem de Máquina

Publications

The contributions of this thesis resulted in the following publications in international peer-reviewed conferences and journals:

- J. D. Pereira, J. R. Campos and M. Vieira, "An Exploratory Study on Machine Learning to Combine Security Vulnerability Alerts from Static Analysis Tools," *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, Natal, Brazil, 2019, pp. 1-10, <https://doi.org/10.1109/LADC48089.2019.8995685>
- J. D'Abruzzo Pereira and M. Vieira, "On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects," *2020 16th European Dependable Computing Conference (EDCC)*, Munich, Germany, 2020, pp. 97-102, <https://doi.org/10.1109/EDCC51268.2020.00025>
- José D'Abruzzo Pereira, Rui Silva, Nuno Antunes, Jorge L. M. Silva, Breno de França, Regina Moraes, and Marco Vieira. 2020. A platform to enable self-adaptive cloud applications using trustworthiness properties. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '20)*. Association for Computing Machinery, New York, NY, USA, 71–77. <https://doi.org/10.1145/3387939.3391608>
- J. D. Pereira, J. R. Campos and M. Vieira, "Machine Learning to Combine Static Analysis Alerts with Software Metrics to Detect Security Vulnerabilities: An Empirical Study," *2021 17th European Dependable Computing Conference (EDCC)*, Munich, Germany, 2021, pp. 1-8, <https://doi.org/10.1109/EDCC53658.2021.00008> (*Best paper award*)
- J. D. Pereira, N. Ivaki and M. Vieira, "Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects," in *IEEE Access*, vol. 9, pp. 142879-142892, 2021, <https://doi.org/10.1109/ACCESS.2021.3120349>
- J. D. Pereira, J. H. Antunes and M. Vieira, "A Software Vulnerability Dataset of Large Open Source C/C++ Projects," *2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC)*, Beijing, China, 2022, pp. 152-163, <https://doi.org/10.1109/PRDC55274.2022.00029>
- José D'Abruzzo Pereira, Nuno Lourenço, and Marco Vieira. 2023. On the Use of Deep Graph CNN to Detect Vulnerable C Functions. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing (LADC '22)*. Association for Computing Machinery, New York, NY, USA, 45–50. <https://doi.org/10.1145/3569902.3569913>
- J. D'Abruzzo Pereira and M. Vieira, 2023. An Approach to Characterize the Security of Open-Source Functions using LSP. In *Proceeding of the IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, Florence, Italy, 2023, 137-147. <https://doi.org/10.1109/ISSRE59848.2023.00073>

Two additional publications with preliminary results were published during this Ph.D. work:

- J. D. Pereira, "Techniques and Tools for Advanced Software Vulnerability Detection," *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Coimbra, Portugal, 2020, pp. 123-126, <https://doi.org/10.1109/ISSREW51248.2020.00049> (*Student Forum Paper*)
- J. D. Pereira, J. H. Antunes and M. Vieira, "On Building a Vulnerability Dataset with Static Information from the Source Code," *2021 10th Latin-American Symposium on Dependable Computing (LADC)*, Florianópolis, Brazil, 2021, pp. 1-2, <https://doi.org/10.1109/LADC53747.2021.9672589> (*Workshop Paper*)

The following papers are also related to this thesis, but not included:

- Sara Ferreira Dinis e Silva, José D'Abruzzo Pereira. 2021. A study on software engineering team dysfunctions: an experience report. *Inforum 2021*. (*Communication paper*)
- João David Ribeiro, José D'Abruzzo Pereira, Nuno Antunes. 2022. An Experimental Study of Elasticity in Kubernetes HPA for Microservice Applications. *Inforum 2022*
- José D'Abruzzo Pereira, João David Ribeiro, João Pires, Pedro Moita, Nuno Laranjeiro, and Marco Vieira. 2023. On the use of the TMA Framework to promote self-adaptation capabilities in TalkConnect. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing (LADC '22)*. Association for Computing Machinery, New York, NY, USA, 89–90. <https://doi.org/10.1145/3569902.3569953> (*Fast Abstract*)
- E. Rodrigues, J. D. Pereira and L. Montecchi, "A Model-Driven Approach for the Management and Enforcement of Coding Conventions," in *IEEE Access*, vol. 11, pp. 25735-25754, 2023, <https://doi.org/10.1109/ACCESS.2023.3256886>
- João Rafael Henriques, José D'Abruzzo Pereira. 2023 "On the Automation of Software Vulnerability Collection of a Database with Static Information". *INForum 2023*, pp. 195-198

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Contributions	5
1.3	Outline of the Thesis	7
2	Background and Related Work	9
2.1	Background Concepts	9
2.1.1	From Dependability to Security	9
2.1.2	Software Vulnerabilities	10
2.1.3	Software Development Life Cycle	13
2.1.4	Common Vulnerability Scoring System	14
2.1.5	Classification Performance Metrics	15
2.2	Vulnerability Detection Techniques	17
2.2.1	Static Analysis for Vulnerability Detection	18
2.2.2	Penetration Testing and Fuzzing	21
2.2.3	Software Metrics as Indicators of Vulnerabilities	22
2.2.4	Other Vulnerability Detection Techniques	25
2.3	Related Work	26
2.3.1	Evaluation of Vulnerability Detection Tools	26
2.3.2	Vulnerability Detection and Analysis	28
2.3.3	Vulnerability Datasets	31
2.3.4	ML for Software Security	32
2.3.5	Vulnerability Characterization and Prioritization	35
2.4	Summary	37
3	Building a Vulnerability Dataset	39
3.1	Process to Build the Dataset	40
3.1.1	Collect Vulnerabilities from Online Platforms	40
3.1.2	Retrieve Source Files from Version Control Systems	43
3.1.3	Generate SAT Alerts and Software Metrics	43
3.1.4	Store the Data in a Database	46
3.2	Dataset Characterization	46
3.3	Threats to Validity	49
3.4	Summary	50
4	Evaluating Static Analysis Tools in Large Projects	51
4.1	Vulnerabilities and Approach	52
4.2	Results and Discussion	54
4.2.1	Overall Observations	54

4.2.2	Performance of the Tools	57
4.2.3	Performance per Vulnerability Category	57
4.3	Threats to Validity	60
4.4	Summary	60
5	Understanding Buffer Overflow Vulnerabilities	63
5.1	Vulnerabilities and Approach	64
5.1.1	Select Software Projects	65
5.1.2	Retrieve Vulnerability Metadata	66
5.1.3	Collect Source Code Versions	66
5.1.4	Classify Vulnerabilities using ODC	66
5.1.5	Collect and Analyze SAT Alerts	68
5.1.6	Collect and Analyze Software Metrics	68
5.2	Results and Discussion	68
5.2.1	Main Code Changes when Fixing Vulnerabilities	69
5.2.2	SAT Alerts Before and After Vulnerability Fixes	72
5.2.3	Impact of Vulnerability Fixes on Software Metrics	76
5.3	Threats to Validity	78
5.4	Summary	79
6	Detecting Software Vulnerabilities with Machine Learning	81
6.1	Classical Machine Learning	82
6.1.1	Vulnerabilities and Approach	82
6.1.2	Results and Discussion	85
6.1.3	Understanding the Classification Results	90
6.2	Deep Learning	92
6.2.1	Vulnerabilities and Approach	92
6.2.2	Results and Discussion	96
6.3	Threats to Validity	99
6.4	Summary	100
7	Characterizing Code Units with Logic Scoring of Preference	103
7.1	SCOLP Methodology	104
7.1.1	Quality Model Definition	104
7.1.2	Attribute Extraction and Score Calculation	108
7.1.3	Map Code Units into Priority Groups	109
7.1.4	Combine Priority Groups into a Unified Categorization	109
7.2	Quality Model (QM) Instances	110
7.3	Experimental Evaluation	112
7.3.1	Vulnerabilities and Approach	113
7.3.2	Results and Discussion	114
7.4	Expert Ranking and Validation	119
7.4.1	Profile of the Experts	120
7.4.2	Questionnaire to Validate the Results	120
7.4.3	Experts <i>vs</i> SCOLP	123
7.5	Threats to Validity	124
7.6	Summary	125

8	Framework to Promote Self-Adaptation	127
8.1	Concepts on Self-Adaptive Systems	128
8.2	TMA Platform Architecture	129
8.2.1	Monitor Component	131
8.2.2	Analyze Component	132
8.2.3	Planning Component	133
8.2.4	Execute Component	134
8.2.5	Knowledge	135
8.3	Usage Scenario: Scaling Containers	136
8.3.1	Resource Consumption per Pod Quality Model	138
8.3.2	Performance Quality Model	139
8.4	Results and Discussion	140
8.5	Summary	143
9	Conclusions and Future Work	145
9.1	Conclusions	145
9.2	Future Work	146
	References	149
A	ML to Combine Security Vulnerability Alerts from SATs	171
B	Software Metrics	191

Acronyms

1ooN 1-out-of-N

AC Attack Complexity

AI Artificial Intelligence

AR Availability Requirement

AST Abstract Syntax Tree

AV Attack Vector

AVI Availability Impact

CD Continuous Deployment

CDE Continuous Delivery

CFG Control Flow Graph

CFI Confidentiality Impact

CI Continuous Integration

CM Confusion Matrix

CMS Content Management System

CNN Convolutional Neural Network

CPG Code Property Graph

CR Confidentiality Requirement

CV Cross Validation

CVE Common Vulnerability and Exposures

CVSS Common Vulnerability Scoring System

CWE Common Weakness Enumeration

DDoS Distributed Denial of Service

DFA Data Flow Analysis

DFG Data Flow Graph

DGCNN Deep Graph Convolution Neural Network

DoS Denial of Service

DT Decision Tree

DWA Direct Weight Assessment

ECM Exploit Code Maturity

FCT Foundation for Science and Technology

FIRST Forum of Incident Response and Security Teams

FN False Negative

FNR False Negative Rate

FP False Positive

FPR False Positive Rate

GB Gradient Boosting

GCD Generalized Conjunction/Disjunction

GDPR General Data Protection Regulation

HK Henry Kafura Size

IDE Integrated Development Environment

IDM Importance Decomposition Method

IDS Intrusion Detection System

II Integrity Impact

IoT Internet of Things

IR Integrity Requirement

IRR Inter-Rater Reliability

k-NN k-Nearest Neighbors

LLM Large Language Model

LOC Line of Code

LR Logistic Regression

LSP Logic Scoring of Preference

MCDM Multi-Criteria Decision Making

MDA	Mean Decrease Accuracy
MDG	Mean Decrease Gini
ML	Machine Learning
MTD	Moving Target Defense
NB	Naive Bayes
NIST	National Institute of Standards and Technology
NN	Neural Network
NooN	N-out-of-N
NPV	Negative Predictive Value
NVD	National Vulnerability Database
ODC	Orthogonal Defect Classification
OOP	Object-Oriented Programming
OS	Operating System
OWASP	Open Web Application Security Project
PCA	Principal Component Analysis
PPV	Positive Predictive Value
PR	Privileges Required
PTT	Penetration Testing Tool
QM	Quality Model
RC	Report Confidence
RF	Random Forest
RL	Remediation Level
RQ	Research Question
SAS	Self-Adaptive System
SAT	Static Analysis Tool
SCA	Static Code Analysis
SCOLP	Security Characterization of Open-source functions using Logic Scoring of Preference
SDLC	Software Development Life Cycle

SEI Software Engineering Institute
SM Software Metric
SMOTE Synthetic Minority Over-Sampling Technique
SPT Software Penetration Testing
SQL Structured Query Language
SQLi SQL Injection
SQUARE Security Quality Requirements Engineering
SRM Software Reliability Model
SS Sensitive Sink
SVM Support Vector Machine
TMA Trustworthiness Monitoring & Assessment
TN True Negative
TNR True Negative Rate
TP True Positive
TPR True Positive Rate
UI User Interaction
VDM Vulnerability Discovery Model
VGG Visual Geometry Group
WBR Weight based on Ranking
WPVD WPScan Vulnerability Database
XGB Extreme Gradient Boosting
XP Extreme Programming
XSS Cross-Site Scripting

List of Figures

2.1	Dependability and Security Attributes	10
2.2	Fault model of Vulnerability Discovery Methodology	10
2.3	Metrics used to calculate the CVSS score	15
2.4	Vulnerability Lifecycle	17
2.5	Possible status of a system	18
2.6	Example of a Control Flow Graph	24
3.1	Dataset creation pipeline	41
3.2	Source code repository representation of a vulnerability fix	44
3.3	Analysis of the most common vulnerability types along the years	49
4.1	Number of commits that fixed vulnerabilities	52
4.2	Methodology to run the SATs in the different snapshots source code	54
4.3	Number of patches/commits detected by the SATs	55
4.4	Vulnerable files reported by the SATs with all C/C++ extensions in the relevant directories	56
4.5	Number of alerts per commit along the time (CppCheck)	56
4.6	Number of alerts per commit along the time (Flawfinder)	57
5.1	Approach for analysis of buffer overflow vulnerabilities	65
5.2	Number of changed files per vulnerability fix	70
5.3	Analysis of the buffer overflow vulnerabilities using ODC	70
6.1	Example of the resulting dataset	84
6.2	Histogram with the maximum number of equal software metrics in the vulnerable/neutral files	91
6.3	Approach followed to detect vulnerable functions	93
6.4	An example of a CFG being reduced	94
6.5	Detailed approach of the deep learning classification process	97
6.6	An example of an Adaptive Max Pooling	97
6.7	Confusion Matrix with the prediction for an experiment run	98
7.1	SCOLP approach followed to characterize the code units	105
7.2	Example of a quality model	105
7.3	QM using Software Metrics of volume and complexity	111
7.4	Quality Model with Memory Management (MM) attributes	111
7.5	Quality Model with Input Validation (IV) attributes	111
7.6	Quality Model with Permission (PER) attributes	112
7.7	Workflow to validate if a code unit (function) satisfies the minimum quality criteria	114

7.8	Examples of QM instances with different configuration	116
7.9	Function distribution - Quality Model IV	118
7.10	Function distribution - Quality Model PER	118
7.11	Function distribution - QM IV. Only memory management-related functions considered vulnerable	119
7.12	Function distribution - QM PER. Only memory management- related functions considered vulnerable	120
7.13	Validation process of the Expert Responses	122
8.1	Lifecycle of Trustworthiness Assessment	130
8.2	Architecture and Interfaces of TMA	130
8.3	Resource Consumption QM	132
8.4	Data model used by TMA for the Monitor Component	135
8.5	Data model used by TMA for the Actuators	136
8.6	Example of scaling up the number of pods	137
8.7	Performance QM	139
8.8	Charts of the Resource Consumption and Performance Scores	141
A.1	Overall methodology to combine diverse SATs	176
A.2	Representation of the dataset with samples and features	177
A.3	Venn Diagrams of TPs and FPs for SQLi vulnerabilities	181
A.4	Venn Diagrams of TPs and FPs for XSS vulnerabilities	182
A.5	CM for SQLi vulnerabilities of the prediction (Bagging)	184
A.6	CM for XSS vulnerabilities of the prediction (RF)	184
A.7	Representation of the file dataset with vulnerability number	185
A.8	Proportion of vulnerabilities detected by the Top-N files	187
A.9	Proportion of files detected compared with the Top-N files	187

List of Tables

2.1	The Top 10 list of OWASP from 2021	11
3.1	The vulnerability categories considered for this work and examples of CWE	42
3.2	Examples of SAT rules from Flawfinder	45
3.3	Examples of SAT rules from CppCheck	45
3.4	Summary of the five large C/C++ projects used in our work	47
3.5	Summary of the number of patches and vulnerable code units	47
3.6	The vulnerability categories and the respective number of vulnerabilities.	48
4.1	Vulnerability categories for the Mozilla dataset	53
4.2	SAT Results and Metrics	57
4.3	Results of the SATs - Vulnerability Categories - CppCheck	58
4.4	Results of the SATs - Vulnerability Categories - Flawfinder	58
4.5	Performance of the SATs	59
5.1	Number of Vulnerabilities in the Buffer Overflow Analysis	66
5.2	Vulnerability distribution across the ODC Defect Type and the Qualifiers	70
5.3	Vulnerability distribution across the ODC Defect Type for the projects	71
5.4	Vulnerability distribution across the ODC Qualifier for the projects	71
5.5	Number of alerts reported by the SATs for the complete code-base in all vulnerabilities	73
5.6	CppCheck SAT alerts reported in Linux Kernel that changed from the vulnerable to the neutral versions	73
5.7	CppCheck SAT alerts reported in Mozilla that changed from the vulnerable to the neutral versions	74
5.8	Flawfinder SAT alerts reported in Linux Kernel that changed from the vulnerable to the neutral versions	74
5.9	Flawfinder SAT alerts reported in Mozilla that changed from the vulnerable to the neutral versions	75
5.10	Top 10 SMs impacted by the vulnerability fixes per project	77
5.11	Unchanged SMs in the vulnerability fixes per project	78
6.1	Vulnerabilities and number of fixed files per vulnerability categories in the Mozilla dataset	83
6.2	Algorithms and Techniques	85
6.3	Performance metrics for the binary classification	86

6.4	Performance metrics for the binary classification per category - Memory Management	87
6.5	Performance metrics for the binary classification per category - Input Validation	88
6.6	Performance metrics for the binary classification per category - Permission	88
6.7	Performance metrics per category (multiclass classification)	89
6.8	Features used for the DGCNN+VGG evaluation	95
6.9	Algorithms and Techniques	97
7.1	Attributes overall importance	107
7.2	GGCD.17 operators	108
7.3	Configurations used in the experiments	112
7.4	Summary of the functions per vulnerability category (when vulnerable) of the Linux Kernel project	113
7.5	Configurations used in the experiments	114
7.6	Weights used in the base SM QM	115
7.7	Weights used in the specialized MM QM	115
7.8	Weights used in the specialized IV QM	117
7.9	Weights used in the specialized PER QM	117
7.10	Invited experts by country and affiliation	121
7.11	Description about the selected functions	122
7.12	Priority groups provided by QM instances and by the experts	123
8.1	Slots of demand in the experiment	138
8.2	Average response time and throughput of each configuration	142
A.1	Algorithms and Techniques	178
A.2	Algorithms' Hyperparameters	179
A.3	Results using the 100N diversity heuristic for XSS and SQLi	181
A.4	Metrics for the 100N diversity heuristic for XSS and SQLi	181
A.5	Vulnerabilities identified by the Top-50 files using linear regression and actual number of vulnerabilities	186
B.1	Description of the SMs at the File level	191
B.2	Description of the SMs at the Function level	194
B.3	Description of the SMs at the Class level	195

Chapter 1

Introduction

Most organizations rely on software systems to perform their core activities and functions, frequently dealing with large amounts of money and sensitive data. Such systems are normally connected to the Internet, which increases their exposure [Rizvi et al., 2020; Zhang et al., 2021]. Additionally, software applications are spread on many devices that people interact with on a daily basis [Moore, 2022; Okman, 2022]. Although development teams do their best to deploy software without weaknesses (*e.g.*, relying on detection techniques integrated into the Software Development Life Cycle (SDLC) [Mouratidis et al., 2005]), most applications are released to their end-users with vulnerabilities, a particular type of defect that opens the door to security attacks that can have severe consequences to the organizations and the users [SC27, 2018]. Examples of consequences include unauthorized access, access to confidential information, data breaches, and data integrity violations, ultimately leading to reputation damage, financial losses, and even safety-related violations [Dowd et al., 2006; Neves et al., 2006].

Several vulnerabilities have been discovered in the last few years in widely used software systems, and the consequences are many. For instance, the Log4Shell vulnerability in the widely used Java logging component Apache Log4j2 allows remote code execution [Guardian, 2021]. Because Log4j2 is a Java component widely used in many applications (some of which not maintained anymore), even though this vulnerability has already been fixed, it is still present in some applications, being considered ‘the most critical vulnerability of the last decade’ [Guardian, 2021]. Another example is the Facebook Business Page, in which the attacker could grant himself administrative permission to a Facebook page [Muthiyah, 2015]. In an episode known as the Panama Papers Incident, several documents (including financial information) were leaked in 2015 [Maunder, 2016; of Investigative Journalists, 2016]. The data breach was possible because Mossack Fonseca (the Panama-based law firm whose data was compromised) was using the WordPress Content Management System (CMS) with a vulnerable component named Revolution Slider. These attacks caused several consequences, and the common cause was software vulnerabilities.

Software security is nowadays a major concern, with 26,448 vulnerabilities reported in 2022, an all-time record [Targett, 2023]. This is the consequence of software systems being used in more enterprises and by more people, but that is not

the only reason. In fact, software developers do not always know the best security practices, and although more techniques are being used to identify software vulnerabilities, their effectiveness is disappointing in most scenarios. Additionally, attackers are improving their skills, taking advantage of such security breaches. Consequently, more investment in cybersecurity is being made and is expected to grow, reaching a market value of US\$ 46.3 by 2027 [Brooks, 2023].

Several techniques and tools exist to detect software vulnerabilities. They can be divided in two main groups: (i) *static techniques*, based on the analysis of the source code, and (ii) *dynamic techniques*, in which the software needs to be executed to identify the vulnerabilities. The most known static technique is Static Code Analysis (SCA) [Louridas, 2006]. Static Analysis Tools (SATs) report potential problems (alerts), which need to be analyzed by the software development team to decide if they are an actual problem (*e.g.*, a vulnerability) or not. Such tools can be used by software developers from the early stages of the software development. There are other more recent static techniques that can also be used, for example, based on Machine Learning (ML) [Walden et al., 2014] or genetic algorithms [Medeiros et al., 2017]. In these, static properties are extracted from the source code (*e.g.*, software metrics) and used as input (features) for ML algorithms to predict if an instance (*e.g.*, file or function) is vulnerable or not. On the dynamic techniques, the most known one is Software Penetration Testing (SPT), supported by Penetration Testing Tools (PTTs). Vulnerabilities are detected by simulating attacks in a controlled environment. There are also hybrid approaches, which combine static and dynamic techniques [Amin et al., 2019; Hanif et al., 2021; Kim et al., 2016].

Although several software vulnerability detection techniques do exist, they have key limitations. Tools (*i.e.*, SATs and PTTs) are frequently too expensive for most organizations, but even though they either report a high number of False Positives (FPs) (cases reported that are not actual problems) or a high number of False Negatives (FNs) (unidentified vulnerabilities). As a consequence, development teams are forced to spend a considerable amount of time analyzing the reported cases without being sure that all vulnerabilities have been detected. Hence, **alternative approaches that help development teams increasing the security of their software, either by improving vulnerability detection or by helping the team focusing the effort on the most relevant parts of the code (from a security perspective)**, are urgently needed.

1.1 Problem Statement

Agile Software Development is followed by most software organizations nowadays, with the goal of decreasing the time to market of the systems being developed [digital.ai, 2022]. In fact, agile methodologies (such as Scrum and Kanban) are used by more than 50% of the organizations nowadays [digital.ai, 2022]. DevOps is also a trend [Leite et al., 2019], aiming at breaking the silos between the development team (responsible for the software development) and the operation team (responsible for deploying and monitoring the system in a production en-

vironment), and thus support a smoother and continuous transition between the two [Ebert et al., 2016]. DevOps became popular as a repercussion of the Extreme Programming (XP) agile methodology, which suggests the use of Continuous Integration (CI) as a key practice (requiring new source code to be integrated frequently into the code repository; *e.g.*, GitHub).

The need to shorten the time to market impacts the SDLC. The frequency of deployment to a production environment varies from multiple times a day to once every six months, with most organizations deploying between once per week and once per month [Cloud, 2022]. Every time the source code is pushed to the repository, the CI server starts the build of the source code to check if the code has compiling errors. If there are automated tests, they can be run, and verifications (such as checking the coding standards with the support of a SAT) can be done and reported to the team.

Security related actions can also be conducted to warn the team about potential weaknesses. For instance, software vulnerability detection tools can be automatically triggered by the CI server (*e.g.*, Jenkins, Microsoft Azure DevOps, Atlassian Bamboo, JetBrains TeamCity). These checks are part of a DevSecOps¹ cycle, where a security team is also involved in the DevOps, and security validations are performed as part of the CI build [Prates et al., 2019; Sánchez-Gordón and Colomo-Palacios, 2020]. The sooner the security problems are found, the less expensive it is to fix them and handle the consequences [Chess and McGraw, 2004].

Dynamic vulnerability detection techniques (*e.g.*, penetration testing) usually require a more comprehensive involvement of experts, besides taking considerably more time to execute. Furthermore, a change in a single line of code generates a new version of the software, requiring the complete set of tests to be rerun, and penetration testing tools do not report the exact location of a vulnerability, only that there is a vulnerability that can be exploited via some interface. Static techniques, on the other hand, report the code unit (*i.e.*, file, function, or class) where the potential vulnerability exists. It is also faster to apply static techniques as part of the CI workflow when a change happens, and the analysis can be done only on the modified code units. The focus of this work is on static techniques (and not on the dynamic ones) with the goal of helping development teams improving security from the early stages of the SDLC.

Although SCA can be easily integrated into the DevSecOps lifecycle and is able to report vulnerabilities by scanning the complete source code base (or a part of it) at a low cost, it suffers from three main problems. The first one is the high number of reported FPs, leading software development teams to spend too much time on their analysis (or even giving up on the use of SATs) [Johnson et al., 2013]. The second problem is the FNs, which are actual security problems that are not detected: even if a team analyzes all the reported items, it is not guaranteed that all vulnerabilities in the source code will be found. The third problem is the lack of a complete understanding of how such tools behave in projects with a large source code base, as most studies assess SATs using source code snippets specifi-

¹ Other terms can be used as a synonym of *DevSecOps*, such as *SecDevOps* and *DevOpsSec* [Sánchez-Gordón and Colomo-Palacios, 2020]. The term *DevSecOps* will be used throughout this thesis.

cally crafted for the evaluation. As large software systems are frequently used by a large number of people, not having effective approaches to avoid vulnerabilities leaves the system more prone to attacks with enormous impacts.

Static techniques based on ML suffer from the same issue: either a high number of FPs or FNs [McGraw, 2006; Muske and Khedker, 2015]. Additionally, instead of reporting potentially vulnerable code statements, such techniques usually refer to code units (*i.e.*, files, functions, or classes) [Shin et al., 2011; Walden et al., 2014]. This way, software development teams have to analyze such code units in detail to identify the potential vulnerabilities before fixing them, without being sure if a vulnerability is indeed present in the code unit or not. Consequently, most teams neglect these techniques [Imtiaz et al., 2019]. Nevertheless, given the continuously evolving nature of the ML field, there are clear opportunities for studying its use as support for software security improvement.

The limitations of existing vulnerability detection techniques puts a greater focus on the security efforts of the development team. Due to the limited resources normally available, support is needed to help developers directing the time available to the parts that are more prone to have security issues. This may be done by classifying and ranking the source code [Fischer et al., 2021], but existing techniques mostly focus on previously identified vulnerabilities, making the approach reactive instead of proactive [Morrison et al., 2018; Zeng et al., 2022]. Obviously, the characterization/prioritization of code units can be used to drive further static analysis and testing activities, which may improve vulnerability detection performance.

Datasets with vulnerability-related information are needed to support the development and improvement of vulnerability detection and security improvement techniques. Although several datasets exist [Alves et al., 2016a; Fan et al., 2020; Zheng et al., 2021], these are usually tailored for specific contexts. For instance, they either contain information from synthetic vulnerabilities (*i.e.*, source code that is not part of a real project [SAMATE, 2005]), or do not contain a large number of vulnerabilities [Fan et al., 2020], or are focused in one or a small number of vulnerability types [Li et al., 2018]. Additionally, existing datasets do not include information on how the vulnerability is fixed and the changes in the source code after the fix.

Even when a DevSecOps team puts the best effort on avoiding and detecting software vulnerabilities, some will for sure escape all the security tests and analysis. Mitigating such cases calls for run-time solutions capable of identifying security incidents. For example, as part of the operation activities, an online monitoring mechanism should be deployed to identify and deal with potential issues, either with or without human interaction; *e.g.*, by performing self-adaptation (for self-protection [Yuan et al., 2014] and self-healing [Ghosh et al., 2007]) or implementing Moving Target Defense (MTD) actions [Cai et al., 2016; Cho et al., 2020]. At the very least, the DevSecOps team can be notified to take adequate actions. The problem is that such solutions are either not available to a large audience or are hard to be used.

1.2 Contributions

This thesis advances the state-of-the-art on the **characterization of software code units (e.g., files or functions) from a security vulnerability perspective, making use of static data obtained from the source code**. The proposed techniques can be integrated into the CI pipeline as part of a DevSecOps cycle [Pittet, 2023]. In general, we make use Software Metrics (SMs), alerts reported by the SATs, and information obtained from the Control Flow Graph (CFG) of the code units as input for the proposed techniques. Nevertheless, other static information can be used and integrated into such techniques.

We focus on C/C++ as these languages are the basis for developing widely used systems software (e.g., Operating Systems (OSs) and hypervisors), where the impact of the exploit of a vulnerability may be catastrophic. In fact, according to a White Source report, C and C++ account for 52% of vulnerabilities in open source software (C = 46%; C++ = 6%) [Source, 2021], being *improper use of memory*, which may lead to *buffer overflow*, the most frequent type of vulnerability [Source, 2021].

In short, the main contributions of this thesis are:

- **Vulnerability data collection process and a dataset of vulnerable and non-vulnerable code units** (based on publication [D’Abruzzo Pereira et al., 2022]). The process consists of obtaining information about vulnerabilities of selected open-source projects from online vulnerability databases (e.g., CVE Details [Özkan, 2023]), as well as the vulnerable and the neutral (patched) versions of the source code from the corresponding code repositories. From these two versions of the source code, we collect relevant information, namely Software Metrics (SMs) and alerts reported by SATs. The software vulnerabilities are assigned to categories that reflect the lack of use of the security best practices suggested by the Open Web Application Security Project (OWASP). The dataset currently contains vulnerability data from five C/C++ open-source projects (Mozilla, Linux Kernel, Xen, Apache httpd, and glibc) collected using the SciTools Understand tool [SciTools, 2011] and two open-source SATs (CppCheck [Marjamäki, 2007] and Flawfinder [Wheeler, 2001]). This dataset is used as support for the remaining contributions of this thesis.
- **Analysis of the use of SATs in large C/C++ open-source projects** (based on publication [D’Abruzzo Pereira and Vieira, 2020]). Several vulnerabilities from the Mozilla open-source project are used to study the behavior and performance of two SATs (CppCheck [Marjamäki, 2007] and Flawfinder [Wheeler, 2001]). For each vulnerability, the two versions of the source (vulnerable and fixed/neutral) are used to assess if static analysis could have been used to identify the vulnerabilities before deployment (i.e., to prevent the release of the source code to production with vulnerabilities). Furthermore, we analyze the performance by type of vulnerability. Overall, CppCheck performs better than Flawfinder in terms of the vulnerabilities detected, at the cost of a high number of FPs. In two categories

(Data Protection and Coding Practices), CppCheck could detect most of the vulnerabilities.

- **Buffer overflow vulnerabilities analysis using the Orthogonal Defect Classification (ODC)** (based on publication [D’Abruzzo Pereira et al., 2021]). Software vulnerabilities classified with the MITRE’s CWE-119 identifier (*Improper Restriction of Operations within the Bounds of a Memory Buffer*) identifier [MITRE, 2006b] from three large projects (Linux Kernel, Mozilla, and Xen) are analyzed, using both SMs and SAT alerts. In practice, we study the potential correlation between SMs and buffer overflow vulnerabilities, and the differences in terms of SAT alerts in the vulnerable and neutral versions of code units. Furthermore, the vulnerabilities were classified by two researchers using ODC [Chillarege et al., 1992]. Both the defect type and the qualifier were used in the classification. The main finding is that most buffer overflow vulnerabilities are fixed by adding a check before using a memory space and that fixes typically increase the size and complexity of the source code.
- **Two studies using ML algorithms with static data to predict software vulnerabilities.** In the first study, four classical ML algorithms (Decision Tree (DT), Random Forest (RF), Extreme Gradient Boosting (XGB), and Bagging) are used to predict the presence of vulnerabilities in files of the Mozilla project (based on publication [D’Abruzzo Pereira et al., 2021]). Both SMs and SAT alerts are used as features for the ML algorithms. An analysis of the file characteristics (such as the similarities in the SMs) is also presented. In the second study, we follow an approach based on deep learning with a Deep Graph Convolution Neural Network (DGCNN) and a Visual Geometry Group (VGG) network (based on publication [D’Abruzzo Pereira et al., 2023]). In this case, features extracted from the CFG of each function of the Linux Kernel project are used. These features include attributes related to the CFG structure and memory management. We could observe a high recall in both studies at the cost of a low precision.
- **Security Characterization of Open-source functions using Logic Scoring of Preference (SCOLP) to characterize code units from a security perspective** (based on publication [D’Abruzzo Pereira and Vieira, 2023]). SCOLP categorizes the code units (*e.g.*, files, functions, or classes) using static information from the source code, including SMs and memory management-related attributes extracted from the CFG. Our approach assigns the code units into priority groups, from most critical to least critical. For this, Quality Models (QMs) considering different properties (*e.g.*, memory management, input validation, and permission) were developed, and their output is used to assign the code units into priority groups. Software development teams can integrate SCOLP into the DevSecOps lifecycle and use the priority groups to decide the work. We demonstrate the approach with functions of the Linux Kernel project and rely on the judgment of security experts to validate the outputs. Results show that SCOLP provides categorizations similar to experts without involving experienced personnel.

- **Trustworthiness Monitoring & Assessment (TMA) framework** (see publication [D’Abruzzo Pereira et al., 2020])². TMA brings self-adaptation capabilities to software applications and implements the MAPE-K cycle [IBM, 2006]. A Kubernetes cluster is used to deploy the framework, and each MAPE-K component (Monitor, Analyze, Planning, Execute, and Knowledge) is mapped into a Kubernetes component. TMA relies on QMs to characterize properties of interest regarding the managed system. The integration with the managed system (system target of the adaptation) happens through probes and actuators. Libraries for different programming languages were created and are available to smooth the integration. Due to the nature of the TMA framework, it can be used not only at design-time (e.g., to collect security evidences), but also to detect and mitigate other issues (e.g., intrusions) at run-time. To demonstrate the TMA framework, we present a scenario to scale containers in a microservice application deployed in a Kubernetes cluster³. It is important to mention that SCOLP can be easily integrated within the TMA (Analyze component).

1.3 Outline of the Thesis

The remainder of this thesis is structured as follows.

Chapter 2 provides background on dependability and security concepts. It further details existing software vulnerability detection techniques and other concepts used throughout this thesis. A thorough overview of relevant works in the literature is also presented.

Chapter 3 presents the methodology to collect static information about software vulnerabilities and describes the dataset built. The dataset contains both SMs (for three levels of code units: files, functions, and classes) and alerts generated by two SATs (CppCheck and Flawfinder) for five open-source C/C++ projects (Mozilla, Linux Kernel, Xen, Apache httpd, and glibc).

Chapter 4 presents the study on the use of SATs in large projects. The methodology to evaluate the SATs in vulnerable and neutral versions of the code is presented. SATs are evaluated considering both the ability to detect vulnerable code units and the ability to distinguish vulnerability types (multiclass classification).

Chapter 5 presents an analysis of how buffer overflow vulnerabilities are fixed in large open-source C/C++ projects. The chapter proposes a methodology to collect the vulnerabilities and other relevant information, as well as to classify their fixes. Following, a classification of the vulnerabilities with ODC is presented and discussed, and an analysis of the SMs and SAT alerts is provided. Key observa-

² TMA follows a research direction different from the remaining contributions of this thesis. The approach and the platform were designed and developed prior to some of the other contributions and independently from those. Nevertheless, we consider that it can be applied in a security context, especially in a close and smooth integration with SCOLP.

³ The TMA platform has been used in the context of the ATMOSPHERE project for monitoring security policies and providing a hierarchy of scores that allow the security team deciding when and where to act.

tions for software developers to avoid buffer overflow vulnerabilities in C/C++ projects are presented.

Chapter 6 presents two studies that use ML to detect software vulnerabilities. The first one uses classical ML algorithms, with SMs and SAT alerts as their features to classify vulnerable files. Both binary and multiclass classification are performed, and the results are presented and discussed. The second one uses deep learning (DGCNN + VGG) to classify vulnerable functions. Features extracted from the CFG structure and memory management-related attributes are used as input of the network. The methodology on how to extract the features from the CFGs of the functions is presented. The classification results are presented and discussed.

Chapter 7 presents the SCOLP approach, which can be used to characterize code units using Logic Scoring of Preference (LSP). Several QM instances, reflecting different properties of code units, are presented. A demonstration of SCOLP is done with functions of the Linux Kernel project. Finally, the chapter presents a characterization of a small set of functions done by security experts, which is then compared with the results of SCOLP.

Chapter 8 details the TMA framework, which is based on the MAPE-K cycle for self-adaptive systems. The TMA architecture is detailed, and a demonstration of its use in the context of a microservice application deployed in Kubernetes is presented. The results are discussed and compared with an autoscaling solution target for Kubernetes applications (Kubernetes HPA [Google, 2014a]).

Chapter 9 closes this thesis by presenting the final remarks and putting forward ideas for future research topics.

Appendix A presents an exploratory study on using ML algorithms to detect software vulnerabilities using alerts from SATs. A dataset with two types of vulnerabilities (SQL Injection (SQLi) and Cross-Site Scripting (XSS)) in PHP components of the WordPress CMS is used. *Appendix B* presents the complete list of Software Metrics (SMs) used in the experiments and stored in the dataset.

Chapter 2

Background and Related Work

Advancing the state of the art on software security requires understanding concepts related to dependability, security, and vulnerability detection. Furthermore, throughout the work developed in this thesis, several techniques not directly related to security were also used. This way, this chapter discusses not only security concepts but also introduces other relevant concepts, such as Software Development Life Cycles (SDLCs) and classification performance metrics. Furthermore, the chapter puts in perspective the most relevant related works on vulnerability detection, Static Code Analysis (SCA), Software Penetration Testing (SPT), Software Metrics (SMs), vulnerability datasets, and code characterization/prioritization.

The outline of this chapter is as follows. Then, Section 2.1 presents background concepts. Vulnerability detection techniques are discussed in Section 2.2. Finally, Section 2.3 presents an overview of the most relevant related works.

2.1 Background Concepts

Understanding the contributions in this thesis requires mastering some key concepts on dependability and security, software vulnerabilities, software development life cycles, classification of vulnerabilities, and metrics for evaluating vulnerability detection tools. These are introduced in the following.

2.1.1 From Dependability to Security

Avizienis et al. [2004] define **dependability** as the “ability to avoid service failures that are more frequent and more severe than is acceptable” [Avizienis et al., 2004]. They also define the attributes that compose dependability: 1) *availability*: readiness for correct service; 2) *reliability*: continuity of correct service; 3) *safety*: absence of catastrophic consequences on the user(s) and the environment; 4) *integrity*: absence of improper system alterations; and 5) *maintainability*: ability to undergo modifications and repairs.

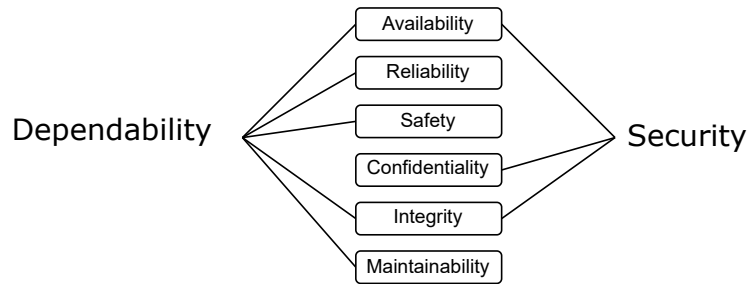


Figure 2.1: Dependability and Security Attributes (adapted from [Avizienis et al., 2004])

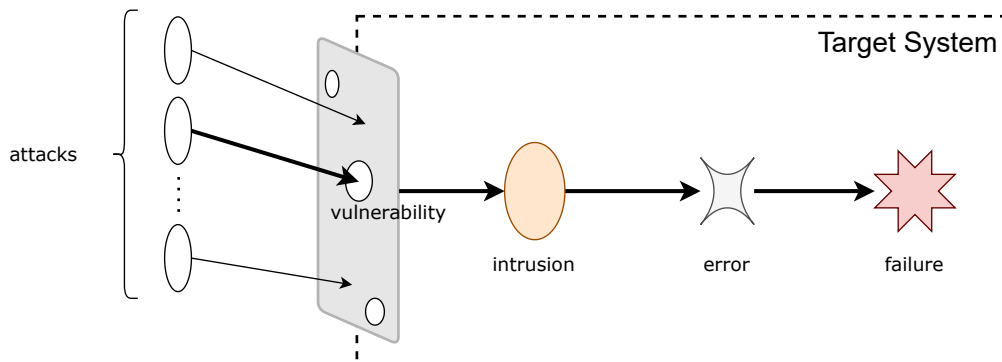


Figure 2.2: Fault model of Vulnerability Discovery Methodology (adapted from [Neves et al., 2006])

Security is “the practice of building software to be secure and function properly under intentional malicious attack” [McGraw, 2006]. Nevertheless, no technique or set of rules will ever be perfect for detecting all security vulnerabilities because security problems evolve over time. Avizienis et al. [2004] also define the attributes of security, which have some overlap with dependability, as shown in Figure 2.1. In addition to *availability* and *integrity*, security includes *confidentiality*. Such attributes can be defined as Avizienis et al. [2004]: *confidentiality* - the absence of unauthorized disclosure of information; *integrity* - the absence of improper system alterations; and *availability* - the readiness for correct service.

2.1.2 Software Vulnerabilities

ISO/IEC 27000:2018 [SC27, 2018] defines *vulnerability* as a “weakness of an asset or control that can be exploited by one or more threats”. According to the Open Web Application Security Project (OWASP) [Foundation, 2021], a vulnerability is a weakness in an application, and it can be caused by “a design flaw or an implementation bug”. When an attack on the application is successful, a vulnerability is activated, resulting in an intrusion (the attacker gains access to the system), as depicted in Figure 2.2. A successful attack may lead to safety violations, data breaches, and financial loss, among other consequences [Neves et al., 2006], depending on the type of vulnerability exploited, the assets, and the intentions of the attacker.

OWASP periodically organizes a list of the Top 10 most critical vulnerabilities in web applications. For that, they group and analyze vulnerabilities based on their CWE identifier. The Common Weakness Enumeration (CWE) [MITRE, 2006a] provides a community-developed list of known software and hardware weaknesses/vulnerabilities for a large set of systems. The last Top 10 list was issued in 2021, and the most critical vulnerabilities are listed in the Table 2.1 [Foundation, 2021]:

Table 2.1: The Top 10 list of OWASP from 2021

OWASP Category	Description
Broken Access Control	When the user acts outside the intended permissions.
Cryptography Failures	Data that must be encrypted is hard-coded or when a risky crypto algorithm is used.
Injection	When data provided by the user is not sanitized before use.
Insecure Design	A new category from 2021 expressed as “missing or ineffective control design”.
Security Misconfiguration	When unnecessary features are installed or enabled.
Vulnerable and Outdated Components	When outdated (potentially) vulnerable components are used.
Identification and Authentication Failures	When brute-force is allowed, or when the software allows weak passwords.
Software and Data Integrity Failures	Related with software updates (usually using Continuous Integration (CI) pipelines) without verifying the integrity.
Security Logging and Monitoring Failures	Related with insufficient security information logging or when sensitive information is included in the log file.
Server-Side Request Forgery (SSRF)	When a web application fetches a remote resource without validating the user-supplied URL.

We discuss some types of frequent vulnerabilities in the following paragraphs.

The third type in the OWASP Top 10 List is *Injection Flaw*, which consists on the possibility of untrusted data being sent to an application with the goal of “executing an unintended command or accessing data without proper authorization”. One of the most common types is *SQL Injection (SQLi)*, but injection vulnerabilities can be of many different types, such as NoSQL injection, Operating System (OS) command injection, and LDAP injection. Injection usually happens due to the lack or improper validation of the input provided by an untrusted source.

Another widely known vulnerability in web applications is *Cross-Site Scripting (XSS)* that occurs when untrusted data is sent through a browser API in order to execute malicious scripts. It usually happens due to the lack of proper validation or escaping in the user-supplied data using a browser API. XSS is found in around two-thirds of all web applications. When exploited, XSS flaws can lead the attacker to execute code remotely. XSS vulnerabilities can be of three types: *i) Reflected XSS, ii) Stored XSS, and iii) DOM-based XSS* [Johns et al., 2008].

Buffer overflow is a weakness related to inadequate memory management. It occurs when the software allows reading or writing in a memory space outside the allocated memory [MITRE, 2006b]. Buffer overflow is not listed as part

of the OWASP Top 10, as most programming languages used in web applications handle memory management properly. On the other hand, C and C++ languages, widely used for the development of systems software, leave the memory management to the developers, and consequently, buffer overflow vulnerabilities may happen. In fact, buffer overflow is the most frequent vulnerability in C/C++ code [Source, 2021]. A typical example of a buffer overflow weakness is the use of the C function `strcpy`, as it copies a string from one memory location to another without checking the boundaries of the destination buffer. As an example, we show below a vulnerable code snippet from the Linux Kernel project (`sound/oss/soundcard.c` before the vulnerability fix CVE-2010-4527). It uses the `strcpy` function in an old version of the Linux Kernel. This vulnerability can be fixed either by replacing the vulnerable function (*i.e.*, `strcpy`) with another function, such as `strncpy`, or by adding a checking before the use of `strcpy`.

```
n = num_mixer_volumes++;

strcpy(mixer_vols[n].name, name);

if (present)
    mixer_vols[n].num = n;
else
    mixer_vols[n].num = -1;
```

In the Common Weakness Enumeration (CWE) categorization system, most memory management-related vulnerabilities are classified under the *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer* identifier [MITRE, 2006b]. Although this is a known problem, it is still the most frequent weakness found in most C/C++ software systems for several reasons: *i)* there is no built-in protection in C/C++ applications against accessing/overwriting data in memory; *ii)* C/C++ software systems do not automatically check whether data written to a buffer respects the bounds of that buffer; and *iii)* several and different possibilities and paths in the code may lead to the buffer overflow issue, which makes it challenging to find all possibilities, especially in complex and large projects.

Among many other resources available, the secure coding practices from the OWASP [Turpin, 2010] and the coding standards from the Software Engineering Institute (SEI) CERT [Software Engineering Institute – Carnegie Mellon University, 2006] provide fundamental guidelines and checklists to handle memory management properly. Despite that, many software systems are still released with buffer overflow vulnerabilities, which are usually caused by the lack of knowledge and experience in applying coding practices and guides throughout the software development cycle.

Although formal approaches to detect vulnerabilities (such as Fagan Inspection [Fagan, 1976]) can be used, automated and experimental approaches are usually preferred. A survey by *Liu et al.* on vulnerability detection details four of the most used techniques [Liu et al., 2012]. SCA is the “evaluation of a system or component without the program execution” [Chess and West, 2007]. *Fuzzing* is a randomized testing technique that generates random character streams for the

tests. *SPT* evaluates the success of simulated attacks of malicious users. *Vulnerability Discovery Models (VDMs)* are based on Software Reliability Models (SRMs), and “specify the general form of the dependence of the vulnerability discovery process on the principal factors that affect it”. Details about vulnerability detection techniques are presented in Section 2.2.

There are also tools to detect intrusions (at run-time), such as Intrusion Detection Systems (IDSs) [Sabahi and Movaghar, 2008]. For instance, a network IDS collects data from the network and analyzes the transmitted packets inside it. Milenkoski et al. [2015] categorize the attack detection methods as: *i)* misuse-based; *ii)* anomaly-based; and *iii)* hybrid-based (use both the misuse-based and anomaly-based). *Misuse* can be detected based on attack signatures (signatures at the database are used to identify intrusions), rules (“if-then” rules are used to characterize computer attacks), or the analysis of state transitions (a finite state machine is deduced to identify intrusions). *Anomaly-based* methods use a baseline profile of the regular network of the system activities to distinguish the regular behavior from anomalous behavior. To do that, regular activities are monitored to build the baseline activity profiles.

2.1.3 Software Development Life Cycle

Software security should be considered throughout the whole SDLC, but not all life cycles have specific phases targeted to security activities. Most times, security is something that organizations address in their tailored processes. In this section, we detail the most used SDLCs nowadays and their integration with security activities.

Agile Software Development is widely used in most software organizations [digital.ai, 2022]. It emerged as an alternative to the waterfall SDLC and other SDLCs that focused on finishing the first phases of the life cycle (requirements, architecture, design, implementation) before moving to validation with the client. The Agile Manifesto [Beck et al., 2001] was published in 2012, focusing on a set of principles. Since then, several agile methodologies/frameworks arose, and the most used nowadays are Scrum and Kanban, by more than 50% of the software organizations [digital.ai, 2022].

Scrum is a framework that defines roles, ceremonies, and artifacts [Schwaber and Beedle, 2001]. The backlog items are organized in a *product backlog*. During a *planning meeting*, a sub-set of items are selected to compose a *sprint backlog*. These are the items that will be developed by the *team* during a *sprint*, a period of two to four weeks for the development. After that, a *potentially shippable product increment* is presented to the *product owner* (client liaison) during the *sprint review* meeting. The *team* discusses process improvements during the *sprint retrospective* meeting, which is usually moderated by the *scrum master* (role responsible for making Scrum to work properly).

Kanban is another popular agile framework used to manage work [Anderson, 2010]. Differently from Scrum, Kanban is less prescriptive and requires only three practices: *i)* visualize the workflow (usually with the kanban board); *ii)* limit the

WIP (work in progress, per state of the backlog items); and *iii*) measure the cycle time (time that it takes to develop a backlog item). No ceremonies or roles are defined by Kanban, leaving this responsibility to the team. The constant completion of backlog items creates new challenges for the software development teams.

DevOps emerges as a solution to solve some of the issues not addressed by other agile methods, and it is the trend nowadays [Leite et al., 2019]. It aims at breaking the silos between development and operation teams, supporting a smoother and continuous transition between the two [Ebert et al., 2016]. As a consequence, CI practices, which gained popularity with the Extreme Programming (XP) agile methodology, are being widely used. Such practices involve building and running automated tests and static analysis every time new source code is pushed into the code repository. Many teams also apply Continuous Delivery (CDE) or Continuous Deployment (CD) practices [Pittet, 2023], in which the new code can be deployed automatically to production after successfully running automated acceptance tests or human acceptance tests. The difference is that CDE deploys to the acceptance environment automatically but not to production, while CD deploys to the production environment automatically [Shahin et al., 2017]. However, while this reduces the burden of deploying a new software version, it may introduce more bugs and vulnerabilities if the CI/CDE/CD pipeline is not properly configured.

DevSecOps brings together security and DevOps [Prates et al., 2019; Sánchez-Gordón and Colomo-Palacios, 2020]. Similarly to DevOps, it requires changing the culture of software development organizations and automating processes. This calls for integrating security checks into the development pipeline to report potential problems or, at least, categorize code units with potential issues and guide the development team in the analysis. In practice, continuous verification techniques can be used to identify problems or test the source code in the CI/CDE/CD pipeline [Duvall et al., 2007; Shahin et al., 2017]. For example, integrating Static Analysis Tools (SATs) and Penetration Testing Tools (PTTs) into the pipeline allows the development team to be continuously warned about potential issues.

Security aspects should be considered in all the phases of a SDLC [Vieira and Antunes, 2013], from requirements elicitation to the decommissioning of the software system. However, most SDLCs do not focus on security or consider it only in some specific phases (*e.g.*, implementation, verification). Although this thesis focuses on approaches to be used only in a concrete phase, it is important to keep in mind the need for *security-by-design* throughout the complete SDLC.

2.1.4 Common Vulnerability Scoring System

Software systems are frequently deployed with software vulnerabilities, and in many situations, it is important to understand the impact of each vulnerability by providing a measurable value for them. The Common Vulnerability Scoring System (CVSS) is a system to provide a score to vulnerabilities [First, 1995]. It is managed by the Forum of Incident Response and Security Teams (FIRST) [FIRST,

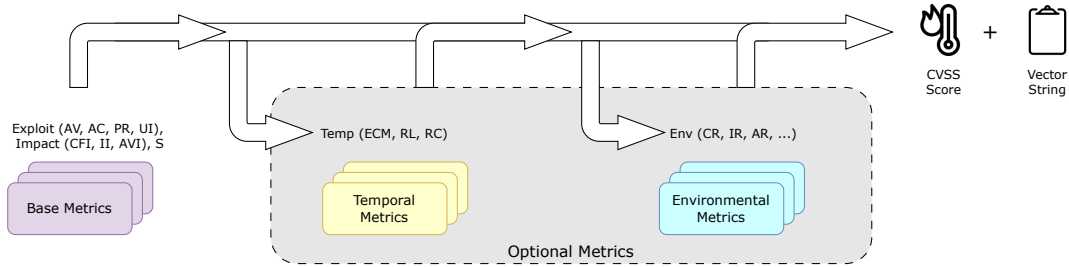


Figure 2.3: Metrics used to calculate the CVSS score (adapted from [FIRST, 2019])

2005], and is currently in version 3.1, released in June 2019. Online databases (e.g., CVE Details [Özkan, 2023] and NVD [of Standards and Technology, 2020]) usually provide the score of a vulnerability using the CVSS.

Three groups of metrics comprises the Common Vulnerability Scoring System (CVSS): i) *base*: intrinsic characteristics of the vulnerability (ranging from 0 to 10); ii) *temporal*: characteristics that change over time; and iii) *environmental*: characteristics unique to the environment of a user. Temporal and Environmental metrics can modify the Base metrics score.

The Base Metrics group is composed of i) Exploitability Metrics (Attack Vector (AV), Attack Complexity (AC), Privileges Required (PR), and User Interaction (UI)); ii) Impact Metrics (Confidentiality Impact (CFI), Integrity Impact (II), and Availability Impact (AVI)); and iii) Scope (S) [Mell et al., 2022]. The Temporal Metrics are composed of Exploit Code Maturity (ECM), Remediation Level (RL), and Report Confidence (RC). Finally, the Environmental Metrics include the Modified Base Metrics and Security Requirement Metrics (Confidentiality Requirement (CR), Integrity Requirement (IR), and Availability Requirement (AR)). A representation of the metrics to calculate the CVSS score can be seen in Figure 2.3.

Spring et al. [2018] criticize the CVSS v3.0. CVSS was created to measure technical severity. However, it is often used for vulnerability prioritization and risk assessment. Thus, one of the criticisms is related to what people want to know, which is the security risk, while CVSS focus on severity. Currently, FIRST is working in CVSS v4.0.

2.1.5 Classification Performance Metrics

To better understand vulnerability detection techniques and tools, we need metrics that support quantitative evaluating (to help us deciding which technique or tool works better depending on the context). Such metrics are also considered in pattern recognition and classification (Machine Learning (ML)) problems, and are used in some chapters of this thesis, such as Chapter 4 and Chapter 6.

In our work, the instances (output of a technique/tool or item being evaluated) are code units (such as files or functions, when evaluating the prediction or classification provided by Machine Learning (ML) algorithms) or alerts reported (by a SAT being tested). Each instance can be classified as:

- i) **True Positive (TP)**: a vulnerable code unit correctly classified or an alert properly issued
- ii) **False Positive (FP)**: a neutral (non-vulnerable) code unit classified as vulnerable or an alert that does not represent a problem
- iii) **True Negative (TN)**: a neutral (non-vulnerable) code unit correctly classified or not identified (alert not reported)
- iv) **False Negative (FN)**: a vulnerable code unit classified as non-vulnerable or not identified (alert not reported)

Using the classification of all items, we can calculate several performance metrics. The following ones are the most known [Alpaydin, 2014]:

Precision (also known as Positive Predictive Value (PPV)): Ratio of TPs among all the identified instances:

$$Precision = \frac{TP}{TP + FP} \quad (2.1)$$

Recall (also known as sensitivity or True Positive Rate (TPR)): ratio of identifying the TPs among all the positive instances:

$$Recall = \frac{TP}{TP + FN} \quad (2.2)$$

F-Measure: harmonic mean of *precision* and *recall*:

$$F - Measure = 2 * \frac{precision * recall}{precision + recall} \quad (2.3)$$

Other metrics are also mentioned in this thesis:

Specificity (also known as selectivity or True Negative Rate (TNR)): Ratio of TNs among all the negative instances:

$$Specificity = \frac{TN}{TN + FP} \quad (2.4)$$

Accuracy: Proportion of correct classifications - both TPs and TNs - among the total number of instances:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (2.5)$$

Bookmaker Informedness: How consistently the classification mechanism predicts the outcome of both TPs and TNs:

$$BookmakerInformedness = TPR + TNR - 1 \quad (2.6)$$

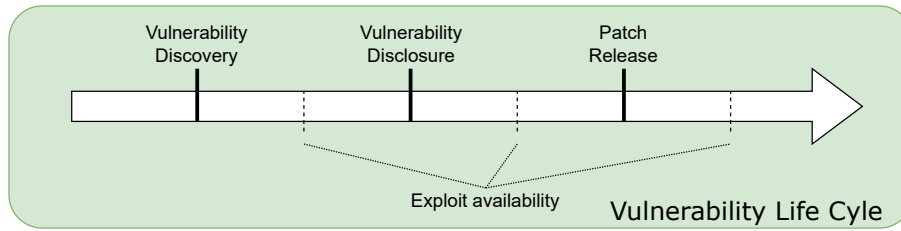


Figure 2.4: Vulnerability Lifecycle (adapted from [Marconato et al., 2012])

Negative Predictive Value (NPV): The proportions of negative results in the classification among the TN results:

$$NPV = \frac{TN}{TN + FN} \quad (2.7)$$

Markedness: How consistently the classification mechanism has the outcome as a marker (*i.e.*, how marked a condition is for the specified classification mechanism, versus chance):

$$Markedness = PPV + NPV - 1 \quad (2.8)$$

2.2 Vulnerability Detection Techniques

During the lifecycle of a vulnerability (shown in Figure 2.4), several events may happen [Marconato et al., 2012]:

- *the discovery of the vulnerability:* the discoverer is aware of the vulnerability and can use it either for a malicious or non-malicious purpose
- *the disclosure of the vulnerability:* the vulnerability is available in a public vulnerability database, such as National Vulnerability Database (NVD) [of Standards and , NIST(2005)]
- *the release of the vulnerability patch:* the system can have the patch applied to remove the vulnerability
- *the availability of the exploit:* the attacker population can exploit the vulnerability

During execution, a system may go through several states [Marconato et al., 2012]: *i) vulnerable:* when a vulnerability is present in the system; *ii) exposed:* when an exploit to the vulnerability is available; *iii) compromised:* when an attack happens to the system and is well-succeeded; *iv) patched:* when the vulnerability is patched, but the system can still be facing some consequences (*e.g.*, the attacker still has administrative privileges); and *v) secure:* when no vulnerabilities are present, and no side-effects happen due to previous attacks. The transitions between the states can be seen in Figure 2.5.

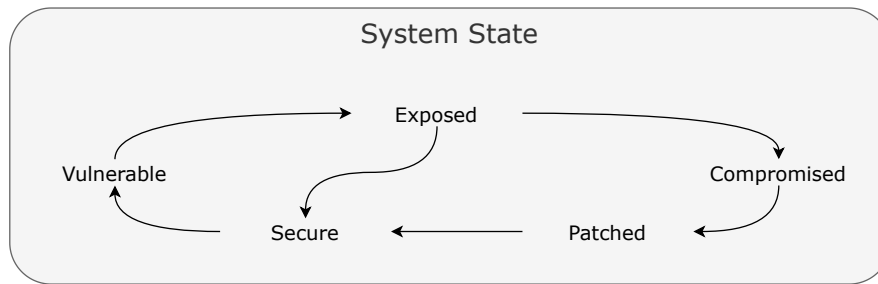


Figure 2.5: Possible status of a system (adapted from [Marconato et al., 2012])

Several techniques allow the discovery of software vulnerabilities [Hanif et al., 2021], which can be divided into (1) conventional techniques, and (2) data mining and machine learning-based techniques. Among the conventional techniques, there are dynamic techniques, static techniques, and hybrid techniques [Hanif et al., 2021; Liu et al., 2012]. Dynamic techniques, such as SPT and Fuzzing [Arkin et al., 2005], involve the execution of the software system and testing it against simulated (or emulated) attacks. When an attack is well-succeeded, a vulnerability is detected and exploited. On the other hand, static techniques try to find potential vulnerabilities by analyzing the source code without executing it [Chess and McGraw, 2004]. Examples of static techniques are SCA, VDM, and white-box testing. Hybrid techniques combine both static and dynamic techniques to discover vulnerabilities.

This thesis focuses on discovering software vulnerabilities and characterizing potentially vulnerable code units (files and functions) using static techniques. Anyway, for completeness, the next sections discuss static techniques (SCA), dynamic techniques (SPT and fuzzing), and the use of SMs (usually with ML algorithms) for vulnerability detection. Other approaches are also introduced.

2.2.1 Static Analysis for Vulnerability Detection

Static Code Analysis (SCA) can be defined as the “evaluation of a system or component without the program execution” [Chess and West, 2007]. Static Analysis Tools (SATs) are used to decrease the time spent on code reviews, which may involve many people and are expensive [Louridas, 2006]. The output of a SAT is a list of issues to be analyzed by the software development team. Each issue identified and reported by a SAT is called an “alert”. An alert includes the following main attributes: *i*) a type (source of the problem), *ii*) a filename, and *iii*) the line of code in which the alert is raised. Other attributes can also be reported, such as the alert severity, category of the vulnerability, CWE, and a message or a description. In practice, each SAT defines some additional attributes to report.

Different types of analysis can be performed by SATs, and the tools can be of any of the following categories:

- **Type Checking:** validates if the value being assigned to a variable corresponds to the variable type. Most compilers and Integrated Development Environments (IDEs) perform type checking when compiling the code

- **Style Checking:** validates if whitespace, naming, deprecated functions, commenting, and program structure are according to the pre-established pattern. Examples of SATs are PMD [PMD, 2004] and Checkstyle [Checkstyle, 2001]
- **Program Understanding:** reasons over the program and are usually integrated within the IDE. It allows finding all the usages of a method or finding a variable declaration. Some companies provide solutions such as CAST Software [CastSoftware, 1990]
- **Program Verification and Property Checking:** allows receiving a body of code to prove that the implementation is according to the specification. It is related to formal verification. Some companies provide solutions such as Escher Technologies [EscherTechnologies, 1997] and Grammatech [GrammaTech, 2018]
- **Bug Finding:** points out the places where the software will not work as the software developer intended. A set of rules is present in the SATs of this type. Examples of SATs are SpotBugs [SpotBugsTeam, 2017] (which is formerly known as FindBugs [Ayewah et al., 2008]), Coverity [Synopsys, 2006], and Klocwork [Software, 2014]
- **Security Review:** identifies security problems. In this case, SATs can be considered as a hybrid of property checkers and bug finders. Examples of SATs are Pixy [Jovanovic et al., 2006], RIPS [Dahse and Holz, 2014], phpSAFE [Nunes et al., 2015], WAP [Medeiros et al., 2014], WeVerca [Hauzar and Kofron, 2015], ITS4 [Viega et al., 2000], CppCheck [Marjamäki, 2007], Flawfinder [Wheeler, 2001], CppTest [Parasoft, 2010], and SonarQube [Campbell and Papapetrou, 2013]

Static Analysis Tool (SAT) normally checks a set of rules specified for a specific programming language to emit the alerts. The alerts can be of diverse types, such as memory errors, resource leaks, violation of APIs or framework rules, exceptions, encapsulation violations, race conditions, and security vulnerabilities. Diverse techniques can be used to identify the alerts [Austin et al., 2013; Freitez et al., 2009; Gosain and Sharma, 2015], including *(i)* syntactic pattern matching, *(ii)* lexical analysis, *(iii)* Data Flow Analysis (and its particular case of taint analysis), *(iv)* parsing, *(v)* model checking, and *(vi)* symbolic execution.

The simplest technique is *syntactic pattern matching* [Chess and McGraw, 2004]. If the source code matches a rule that indicates a problem, then an alert is raised. An example is presented below, where the `switch` statement does not account for all possible values of the enumerated type (adapted from [Chess and West, 2007]):

```
typedef enum { red, green, blue } Color;

char* getColorString(Color c) {
    char* ret = NULL;
    switch (c) {
        case red: printf("red");
    }
    return ret;
}
```

Data Flow Analysis (DFA) is another technique [Ayewah et al., 2008]. SATs use the possible values that variables can have and evaluate if an exception may happen when running the code, such as a null pointer exception. This is done by evaluating all the definitions of a variable, its use to compute other values, and its use in predicates [Horgan et al., 1994]. An example (using Java) can be seen below where the third line raises an exception in case the variable *g* is null (adapted from [Ayewah et al., 2008]):

```
if (g != null)
    paintScrollBars(g, colors);
g.dispose();
```

A special case of DFA is *taint analysis* (also known as *tainted data-flow analysis* [Ghaffarian and Shahriari, 2017]), used for vulnerability detection. In the taint analysis, all data provided by an untrusted source is considered as tainted [Schwartz et al., 2010]. If tainted data reaches a Sensitive Sink (SS), the SAT will report and alert with a vulnerability [Medeiros et al., 2016]. A Sensitive Sink (SS) is a request or the output to another source. For instance, the execution of a Structured Query Language (SQL) query is a SS. The alert will not be issued either if no data provided by another source is used or if the tainted data goes to a sanitization process. An example using Java can be seen below, where the user input (*i.e.*, tainted data) is used in the SQL query without sanitizing the source.

```
data = System.getProperty("user.home");
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.createStatement();
sqlQuery = "insert_into_users_(status)_values_('updated')"
           + "_where_name='"+data+"'";
int rowCount = sqlStatement.executeUpdate(sqlQuery);
IO.writeLine("Updated_" + rowCount +
            "_rows_successfully.");
```

SATs can usually detect this type of vulnerability, but there are some challenging aspects. Two examples are the use of variables that are literals and the use of sanitization functions unknown by the detection tool, which frequently lead to FPs or FNs. An example can be seen in the following source code.


```

data = "foo";
dbConnection = IO.getDBConnection();
sqlStatement = dbConnection.createStatement();
sqlQuery = "insert_into_users_(status)_values_('updated') "
           + "_where_name='" + data + "'";
int rowCount = sqlStatement.executeUpdate(sqlQuery);
IO.writeLine("Updated_" + rowCount +
            "_rows_successfully.");

```

2.2.2 Penetration Testing and Fuzzing

A key problem with security testing is that people focus on validating that the tested features work (positive) instead of focusing on detecting potential situations where they can lead to not working properly (negative). Even when the negative is tested, it can not be said that there are no faults, but it can be said that faults are not discovered under certain conditions/scenarios [Arkin et al., 2005].

Security assessments are usually done at the end of the SDLC, in a time-boxed manner [Vieira and Antunes, 2013]. However, it should be done throughout the whole process, and different assessment techniques can be used to do that. Examples of security assessments include attack surface analysis, red teams, fuzzing, and SPT.

Software Penetration Testing (SPT) is a specialization of robustness testing, which consists of submitting the program to exceptional data values [Kropp et al., 1998]. In addition to valid inputs, in which the program should work properly, invalid inputs are used to validate whether or not the program responds with a reproducible failure. In both testing approaches, the tester does not need to know the implementation details. Unlike robustness testing, the goal of SPT is to stress the application from the perspective of the attacker in order to reveal vulnerabilities.

Similarly to SCA, SPT should be supported by tools, which are called PTTs. They allow an application to be tested automatically for vulnerabilities based on tampered input values. Examples of PTTs are IBM AppScan [IBM, 2019] and Acunetix [Acunetix, 2019]. SPT should be done prior to system integration and should start at the feature, component, or unit level. When a vulnerability is discovered, developers should also do a root-cause analysis. However, this is not always done in many cases.

An example of a vulnerability that can be detected through SPT is SQLi. When a SQLi vulnerability is exploited, an SQL statement is altered, and an attacker can read or modify the database data. An example of SQL construct with a SQLi can be seen below:

```

String sql = "SELECT * FROM users WHERE " +
            "username='" + email + "' AND " +
            "password='" + password + "'";

```

If the attacker uses the string ' OR 1=1 - instead of an actual email, the expression will be evaluated as true due to 1 = 1 comparison, and it becomes a tautology. As a consequence, the attacker will gain access to the system. Using SPT, the tester tries to simulate an attack by using malicious inputs.

Fuzzing (also known as random-testing [Ghaffarian and Shahriari, 2017]) is another dynamic security testing approach [Bekrar et al., 2011]. Random or invalid values are injected into a program to identify errors and potential vulnerabilities. Such values are randomly mutated from well-formed input data [Ghaffarian and Shahriari, 2017]. Three different methods can be used [Beaman et al., 2022]:

1. **Blackbox fuzzing:** correct data are used to generate random values. No previous knowledge of the application (such as the code execution path) is used to help guide the fuzzer tool
2. **Whitebox fuzzing:** complete knowledge of the application code and behavior is assumed. To increase the code coverage, techniques such as symbolic execution and program analysis are used to guide the fuzzer through the code execution paths
3. **Greybox fuzzing:** between Blackbox and Whitebox fuzzing, and takes advantage of both. However, even though techniques such as program analysis are used, the fuzzer is limited compared to whitebox fuzzing

The following code presents an example of a challenging situation for SPTs. As the method is void, no value is returned to the invocator of the function (*e.g.*, the testing tool). Also, the exception related to SQL malformation is neither handled nor thrown to the invocator.

```
public void operation(String str) {
    try {
        String sql = "DELETE_FROM_table" +
            "WHERE_id='" + str + "'";
        statement.executeUpdate(sql);
    } catch (SQLException se) { }
}
```

2.2.3 Software Metrics as Indicators of Vulnerabilities

Software Metrics (SMs) are frequently used as indicators of the quality and maintainability of the source code [Gaffney, 1981]. Ghaffarian and Shahriari [2017] define this technique as vulnerability prediction models. From the systematic literature review performed by Heckman and Williams [2011], most of the studies use code characteristics as input to automate the Static Code Analysis (SCA). They result from the metrics of the source code.

A classic set of metrics for Object-Oriented Programming (OOP) Paradigm was proposed by [Chidamber and Kemerer, 1991], being later improved in [Chidamber and Kemerer, 1994]. They are commonly known as "Metrics CK" due

to the authors that proposed them (Chidamber and Kemerer), and the set contains six metrics:

- **Weighted Methods per Class (WMC):** the sum of the complexity of all the methods of a class
- **Depth of Inheritance Tree (DIT):** maximum length from one (class) node to the root of the tree (most primitive class);
- **Number of Children (NOC):** number of sub-classes subordinated to a class in the class hierarchy
- **Coupling Between Object Classes (CBO):** count of the number of other classes to which a class is coupled
- **Response For a Class (RFC):** cardinality of the response set of a class
- **Lack of Cohesion in Methods (LCOM):** the difference between the pairs of methods that are similar and the pairs of methods that are not similar

One of the first metrics to define complexity is Cyclomatic Complexity, defined by Thomas J. McCabe in [McCabe, 1976]. This metric is calculated through the Control Flow Graph (CFG) of a program, and the number of feasible paths in the graph represents the cyclomatic complexity. The following formula can also be used to calculate the cyclomatic complexity C :

$$C = E - N + 2$$

where E is the number of edges in the graph, and N is the number of nodes in the graph.

The cyclomatic complexity was initially defined to help deciding the number of test cases to be run when testing a program. Another way of calculating it is by counting the number of regions that a CFG has. An example can be seen on Figure 2.6, in which five closed regions of the graph are represented.

Halstead [1977] defines complexity metrics. His goal was to establish metrics that could relate to each other, such as physical properties. Some of the metrics are related to the operators, operands, the size of a program, its vocabulary, the length, the difficulty, and the effort. Shen et al. [1983] publishes another paper criticizing Halstead's theory of Software Science. Nevertheless, his metrics have been used in the security context [Al-Far et al., 2018; Davari and Zulkernine, 2016] and to predict software vulnerabilities [Chen et al., 2020; Viszkok et al., 2021].

SMs can be classified into different categories:

- **Complexity:** indicate how complex the code unit is, *e.g.* cyclomatic complexity
- **Volume:** characterize the size of the code unit, *e.g.* lines of code

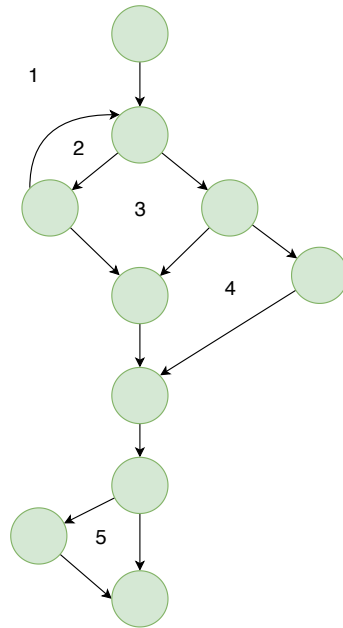


Figure 2.6: Example of a Control Flow Graph (adapted from [McCabe, 1976])

- **Coupling:** indicate how coupled or not the code unit is with other units, *e.g.* coupling between objects
- **Cohesion:** characterize how cohesive or not the code unit is, *e.g.* lack of cohesion

Different tools can be used to calculate SMs. An example is SciTools Understand [SciTools, 2011]. The metrics are divided into three categories (complexity metrics, volume metrics, and object-oriented metrics). SciTools Understand supports many programming languages including Java, C/C++, C#, and Python, among others.

Other tools are also available, such as Eclipse Metrics Plugin (<http://metrics.sourceforge.net/>), CodeMR (<https://www.codemr.co.uk/>). Another widely used tool is SonarQube (<https://www.sonarqube.org/>), which is a tool whose goal is continuous inspection. It aggregates rules for different programming languages to detect bugs, code smells, and vulnerabilities, in addition to SMs.

Previous studies have used SMs as indicators of vulnerabilities. Ghaffarian and Shahriari [2017] designate the works using this technique as *vulnerability prediction model based on Software Metrics (SMs)*. For instance, Chowdhury and Zulkerine [2010] validate the correlation of complexity, coupling and cohesion metrics with vulnerabilities using statistical approaches. Medeiros et al. [2017] use genetic algorithms to combine SMs and predict vulnerabilities, while Alves et al. [2016b] use ML algorithms from a previously classified database of vulnerabilities. See Sections 2.3.4 and 2.3.5 for more details on related work.

2.2.4 Other Vulnerability Detection Techniques

Other characteristics can be used for vulnerability detection. Ghaffarian and Shahriari [2017] define three categories of techniques:

1. **Anomaly detection:** find patterns that do not conform to the expected behavior. This is done in runtime and consists of a search for abnormal patterns in the historical profile of valid behavior [Antunes and Vieira, 2015]
2. **Vulnerable code pattern recognition:** identify vulnerabilities through the analysis of source code structures, such as Abstract Syntax Trees (ASTs), Code Property Graphs (CPGs), or system calls [Li et al., 2022]
3. **Miscellaneous approaches:** other works that do not constitute a coherent category. Usually, they use different Artificial Intelligence (AI) and data science techniques. For instance, genetic algorithms [Medeiros et al., 2017]

Heckman and Williams [2011] identify different types of features or data that can be used to support vulnerability detection: *i*) Source Code Repository Metrics (e.g., code churn, revision history), *ii*) Bug database metrics (such as mapping the bug change in the source code repository to identify fault fixes), and *iii*) Dynamic Analysis Metrics (dynamic analysis results serving as input to or refinement of SCA).

Austin et al. [2013] state that a “single technique for vulnerability discovery is insufficient for finding all types of vulnerabilities”. The Integrated Software Source Code Analysis (ISA) approach uses data fusion to combine the output of 3 SATs (Rats, Its4, Flawfinder) [Kong et al., 2007]. The analyzed projects are *wu-ftpd*, *Net-tools*, *Pure-ftpd*. A score for each vulnerability is defined based on the score of each tool for each vulnerability and the weight of each tool. The authors compare the FPs and FNs of each tool.

Shin et al. [2011] analyze the ability of three groups of metrics (complexity, code churn, and developer activity metrics) to discriminate vulnerable files from neutral files. Two projects were used in their study: Mozilla and Red Hat Linux Kernel. For both projects, some releases between 2005 and 2008 were considered. The number of vulnerable files was small (always less than 200 files) per release, which contained between 10,000 and 13,500 files. Nevertheless, their results show that 24 of the 28 analyzed metrics have the power to discriminate the vulnerable from the neutral (non-vulnerable) files for both projects.

The attack surface measurement is defined by Manadhata and Wing [2011] and is used as an indicator of the security of the system. It is based on the subset of the resources (methods, channels, and data) that can be potentially used to attack the system. Although it is a good measurement to understand the vulnerability fixes, it can also be a problem since adding new features results in an increment in the attack surface measurement.

The concept of AST can also be used as another source of information [Neamtiu et al., 2005]. An AST is a representation in a tree of the source code created by the

compiler, and it contains the variables assignments and conditions as tree nodes. The compiler also uses it to optimize the code to be run.

Another possible source of information is code smells, a characteristic of the code that may “correspond to a deeper problem in the system” [Fowler, 2006]. Examples of code smells are duplicated code, large code, and long parameter list. Some of the SATs can also detect code smells, but there are also tools specifically with this goal (sometimes integrated with the IDE).

2.3 Related Work

The following sections present works related to the problem addressed in this thesis, including the evaluation of vulnerability detection tools, vulnerability analysis, vulnerability datasets, the use of ML in security, and vulnerability characterization and prioritization.

2.3.1 Evaluation of Vulnerability Detection Tools

Austin et al. [2013] evaluate the effectiveness and efficiency of four vulnerability discovery techniques: exploratory manual penetration testing, systematic manual penetration testing, automated penetration testing, and automated static analysis. They analyze three health-system open-source projects, and eight vulnerability types (such as SQLi, XSS, and command injection) are identified. SCA is the most effective technique, although not the most efficient as the False Positive Rate (FPR) is 98%.

Nong et al. [2021] evaluate vulnerability detection tools (both static and dynamic analysis tools) to identify memory-related vulnerabilities. They use the dataset created by Toyota, with 638 test cases, and consider one SAT (*i.e.*, CBMC) and four dynamic analysis tools (*i.e.*, AddressSanitizer, Valgrind, MemorySanitizer, DrMemory). Their results show that SAT accuracy needs to be improved and that it is difficult to obtain both good precision and recall for the same tool. Moreover, tools that use a hybrid approach (static and dynamic techniques) usually have better accuracy. Due to the limitations of the dataset selected, it may not be representative of vulnerabilities in real projects.

Antunes and Vieira [2009b] compared the effectiveness of three commercial PTTs and one developed in a previous work [Antunes and Vieira, 2009a] with three SATs for SQLi vulnerabilities. The tests are performed in web services, either from TPP-App or from a public code from the internet. Their results show better coverage for SATs. Nevertheless, both SATs and PTTs have a high FPR.

Zheng et al. [2006] evaluate the capability of static techniques to detect faults. Three Nortel projects written in C/C++ are analyzed using the data from three SATs (FlexeLint, Klockwork, and Reasoning’s Illuma) previously included in the Nortel inspection process. They use Orthogonal Defect Classification (ODC) to identify the faults and failure types detected by the three techniques studied

(static analysis, inspection, and testing). The results indicate that testing is two to three times more effective than SCA and inspection, which have similar performance. Additionally, the used SATs are effective at identifying two ODC defect types (assignment and checking). Their results also indicate that SATs can be used to find vulnerabilities caused by programming errors. However, their findings are limited to the projects of a single organization (Nortel) and may not be valid in other contexts.

Arusoae et al. [2017] benchmark 11 distinct open source C/C++ SATs using 638 test cases from a test suite created by Toyota [Shiraishi et al., 2015]. Different SAT configurations (Code Sonar, Clang Static Analyzer, CPPCheck, Flawfinder, Flint++, Frama-C, Facebook Infer, Oclint, Sparse, Splint, and Uno) are used. In addition to the three statistics proposed in the original ITC test suite (detection rate (DR), FPR, and productivity), they suggested the new metric called robust detection rate (RDR). Their results show that Oclint has the best detection rate and the highest FPR. Also, Frama-C has the best productivity value and the highest robust detection rate. Although the test suite includes an extensive number of test cases, they are not from real projects. Their goal is to benchmark the SATs using a well-defined set of vulnerabilities that can potentially be detected by the tools, and the work does not analyze the vulnerabilities that could not be detected. Nevertheless, results clearly show that SATs have great limitations.

Braga et al. [2019] study the use of SATs to identify the incorrect use of cryptography functions. The evaluated SATs (FindBugs/FindSecBugs, Xanitizer, SonarQube/sonar-scanner, VisualCodeGrepper, and Yasca) presented a low recall (0.337 or less). Nevertheless, they believe that SATs can help in the development of cryptography software. Their results show that most of the cryptography misuses are not identified by any evaluated SATs. Also, the identified misuses are detected by a small number of SATs.

Harzevili et al. [2023] evaluate the ability of SATs to detect vulnerabilities in ML libraries. The authors used the following SATs: Flawfinder, RATS, CppCheck, Facebook Infer, and Clang static analyzer. Four ML libraries were used in their evaluation: Mlpack, MXNet, PyTorch, and TensorFlow. The tools could identify only 5 of the 410 known vulnerabilities of these libraries. Their results show that Flawfinder could achieve a TPR of 4.95%, and RATS 3.07%. The other three tools could not detect any vulnerability (TPR = 0%). The authors also suggested improvements for the SAT to address the weaknesses of the ML libraries, such as handling the macros allowed by the C programming language.

Filus and Domańska [2023] study the ability of three SATs (CppCheck, Flawfinder, and Visual Code Grepper (VCG)) to detect vulnerabilities in the TensorFlow, a platform used in ML. They focused on vulnerabilities reported in CVE Details and with a CWE that belongs to the CWE Top 25 Most Dangerous Software Weaknesses [MITRE, 2021]. Their dataset contains 104 vulnerabilities of six CWEs (CWE-125, CWE-476, CWE-787, CWE-20, CWE-119, CWE-190) related to memory management issues, integer overflow, improper input validation, and NULL point dereference. The authors used the ODC, and they realized that a missing or incorrect checking statement causes most vulnerabilities. The CVSS scores were analyzed, and the ones with the highest values are the ones related to

memory-related issues (*CWE-787*, *CWE-119*, and *CWE-125*). The SATs were also run in the vulnerable version of the code and the neutral (after the vulnerability is removed). Flawfinder and VCG decreased by two and one alerts, respectively, in the neutral version compared to the vulnerable version for the *CWE-125*. For the *CWE-190*, VCG in the neutral version reported an extra alert, which is a FP. CppCheck had exactly the same number of alerts in the vulnerable and neutral versions.

Antunes and Vieira [2015] proposed a benchmark to evaluate vulnerability detection tools. They use three SATs, four penetration testing tools, and an anomaly detector to detect SQLi vulnerabilities. The workloads for the benchmark are based on a set of web services, although the user can define the workload. Results show that the benchmark can effectively rank tools.

Nunes et al. [2018] created a benchmark for evaluating SATs for Web Security. It includes a workload with vulnerabilities of 134 WordPress plugins, and four scenarios are considered, ranging from business-critical applications to lower-quality applications. They focus on SQLi and XSS vulnerabilities since they are some of the most critical web application vulnerabilities. They use five SATs (phpSAFE, RIPS, WAP, Pixy, and WeVerca), which were ranked during the benchmarking campaign.

Martínez et al. [2022] created an approach to benchmark security tools. It follows a Multi-Criteria Decision Making (MCDM) approach, which relies on expert judgment. Each expert was requested to compare metrics (recall, precision, F-measure, informedness, and markedness) for four scenarios (business-critical, heightened-critical, best-effort, and minimum effort). They performed case studies using vulnerability detection tools and IDS tools to validate their approach. The authors needed to ensure that all answers were consistent because the comparison was pairwise between all the metrics for each scenario. Hence, they calculated the consistency ratio (CR), which was used to discard inconsistent answers. Using the remaining answers, they could prioritize the tools.

2.3.2 Vulnerability Detection and Analysis

Algaith et al. [2018] combine the results of different SATs to decrease the FNs. The results are obtained from the alerts of five SATs (namely phpSAFE, RIPS, WAP, Pixy, and WeVerca) of PHP plugins of WordPress Content Management System (CMS). Three heuristics used to combine the SATs are proposed to identify SQLi and XSS vulnerabilities: *i*) 1-out-of-N (1ooN): raises the alert when any of the SATs report the alert; *ii*) N-out-of-N (NooN): raises the alert when all of the SATs report the alert; and *iii*) simple majority: raises the alert when the simple majority of the SATs reports the alert. Results show that NooN has a better specificity than 1ooN or *simple majority*. 1ooN leads to a better detection (*i.e.* higher recall), but also large number of FPs (*i.e.* low precision).

Meng et al. [2008] use a data fusion approach, and the following SATs are used: FindBugs [Ayewah et al., 2008], PMD [PMD, 2004], JLint [JLint, 2002]. The output of the different tools is presented in a single report. Two policies are used to

prioritize the results: i) using the alert severity, and ii) using the count of tools that produced alerts of that type for that place in the code.

Sampaio and Garcia [2016] use an early vulnerability detection approach. They reason that SCA is performed late in the SDLC, and they suggest detecting vulnerabilities during the development. Context-sensitive Data Flow Analysis (DFA) is used for improving vulnerability detection and mitigating the limitations of pattern matching. They notice that Context-sensitive DFA reduces the number of FPs and early vulnerability detection improves the awareness of the developers about the vulnerabilities being introduced. A plugin for detecting vulnerabilities in Java code was created for Eclipse IDE. They analyze the results using precision, recall, and F-Measure.

Balzarotti et al. [2008] present an approach that combines static and dynamic analysis to identify wrong sanitization procedures. Sanitization is the process of removing possible malicious elements from the input to avoid an attack. The developed tool is called Saner, and it analyzes PHP applications. It is based on the vulnerability scanner Pixy [Jovanovic et al., 2006]. Their process involves creating a sanitization graph and labeling the variables as tainted/untainted to report the vulnerabilities. Five PHP applications are used to validate their methodology. From the experiments, they noticed that sanitization mechanisms used in real-world applications are not always effective and can be bypassed by some attackers.

Shin et al. [2006] propose an approach that combines static and dynamic analysis to detect SQLi vulnerabilities, and the prototype tool is SQLUnitGen. Two tools, AMNESIA and JCrasher, are the base for building SQLUnitGen. While the former creates an automata using SCA that will allow running dynamically defined queries, the latter generates test cases with predefined input values using JUnit. Two small web applications are used to evaluate their approach, which has no FPs and a small number of FNs. The results are compared to FindBugs [Ayewah et al., 2008].

Peng et al. [2023] analyzed Text-To-SQL models of six commercial applications (Baidu-Unit, ChatGPT, AI2SQL, AiHelperBot, Text2SQL, and ToolSKE) that can be used to produce SQL queries from human language. However, the generated SQL queries can be prone to SQLi vulnerabilities. In the evaluation performed by the authors, it was noticed that Baidu-Unit received a payload for a Denial of Service (DoS) attack, and one of the nodes became unavailable, revealing that this tool can be potentially impacted by a Distributed Denial of Service (DDoS) attack. Four tools (ChatGPT, AiHelperBot, Text2SQL, and ToolSKE) can be prone to simple in-band injection attacks, a specialization of a SQLi attack in which the response is received by the same communication channel as the expected behavior. The authors also highlighted some risk mitigation strategies, such as examining if the inputs contain suspicious strings (to avoid black-box attacks), double-check if the model supplier is trustworthy (against backdoor attacks), and use good software engineering practices (*e.g.*, using the Principle of the Least Privilege, and maintain regular database backups). Such tools should not allow words that are reserved by the language (such as drop) and should use the "Prepared Statement technique" [Thomas et al., 2009] to generate the queries.

Jia et al. [2017] propose an offline analysis solution called HOTrace to identify heap vulnerabilities. To do that, HOTrace uses execution traces and identifies taint attributes during the execution. The whole process is done in the programs themselves, without the source code. The evaluation was performed in 17 Windows x86/x64 applications. Using their prototype, they identified 47 previously unknown heap overflow vulnerabilities, including two vulnerabilities in Microsoft Word.

Haller et al. [2013] created the fuzzer Dowser to detect buffer overflow violations. To do that, they combine taint tracking, program analysis, and symbolic execution. Dowser ranks the source code to perform the analysis, which is done based on the data-flow graph. It uses the intuition that complex control flows are more prone to buffer overflow vulnerabilities. They also reduce the symbolic input using dynamic taint analysis to improve the performance of Dowser. The evaluation was performed in six applications (nginx, ffmpeg, inspircd, libexif, poppler, and snort). They could identify two previously unknown buffer overflow vulnerabilities.

Liu et al. [2020] analyzed five open-source C/C++ projects (Linux Kernel, FFmpeg, ImageMagick, OpenSSL, and php-src) and presented 12 findings, which were applied to find 10 zero-day vulnerabilities. The authors wanted to understand if more vulnerabilities can be found close to identified vulnerabilities. Using the commits to fix the identified vulnerabilities, they built the call-graph for the vulnerable code snippets. They found that the vulnerabilities usually follow the Pareto law and that 60% of the vulnerabilities have at least another one close to them (in a 2-jump range in the call-graph).

Morrison et al. [2018] created ODC+V, which is an extension of ODC tailored to classify software vulnerabilities. As the standard ODC has a single value, “security/integrity”, for the “impact” attribute, authors claim that it is not possible to characterize the impact of vulnerabilities in a precise way. ODC+V proposes a new attribute called “security impact”, which may have one of the following values from Microsoft STRIDE [Shostack, 2014]: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. The work presents an evaluation considering 583 defects and 583 vulnerabilities from three projects (Firefox, PHPMyAdmin, and Chrome). The goal is to compare if defects and vulnerabilities are fixed and discovered in the same manner. Results show that vulnerabilities are usually found later in the SDLC compared to “classical” defects. Moreover, vulnerabilities are primarily classified in one of the following ODC defect types: Checking, Assignment/Initialization, or Algorithm/Method. Furthermore, vulnerabilities are often fixed by adding a checking condition (such as an if-clause) rather than other defects. The use of STRIDE in the “impact” attribute has a side-effect: ODC+V is not orthogonal (like the standard ODC) as each attribute may have more than one value.

2.3.3 Vulnerability Datasets

Hanif et al. [2021] presents a taxonomy for software vulnerability detection and the main ML approaches used in the detection process. The authors propose a survey in which they analyzed 90 works of software vulnerability detection from 2011 to 2020. From the analyzed papers, supervised learning and deep learning approaches are a trend, and the types of vulnerabilities most analyzed are buffer overflow, SQLi, and XSS. The vulnerability detection techniques can be conventional (*e.g.*, static analysis, dynamic analysis, or hybrid analysis) or related to data mining and ML (*e.g.*, based on SMs, anomaly detection, or vulnerable code pattern recognition). Nevertheless, most studies suffer from the same issue: they usually have a high number of FNs (unidentified vulnerabilities) or a high number of FPs. The authors show that most works focus on vulnerability detection, and a few are on code and dataset problems. Some of the open issues highlighted by the authors include the need for gold-standard vulnerability datasets and works focusing on multi-vulnerability detection, as most works focus on the most common vulnerability types.

Alves et al. [2016a] developed a dataset with five C/C++ projects (Mozilla, Linux Kernel, Xen Hypervisor, Apache httpd, and glibc) to explore the relationship between SMs and the source code vulnerabilities. The goal was to determine if SMs can discriminate between vulnerable and neutral (non-vulnerable) code. This study found that some SMs were highly correlated, meaning they contain redundant information and should be reduced to one. As such, functions with vulnerabilities in the past are prone to have more vulnerabilities with a probability 55 times higher than functions that never had vulnerabilities in the past.

Reis and Abreu [2017] build a dataset with vulnerabilities and a method to extract them from GitHub's projects. The dataset contains 682 real security vulnerabilities mined from 248 projects from GitHub and almost 2 million commits. The projects belong to several programming languages of the top 5 most popular programming languages on GitHub: JavaScript, Java, Python, Ruby, and PHP. To identify the vulnerabilities, the commit messages are automatically analyzed to look for patterns. Then, a manual diagnosis is performed on the identified commits. Results show that 62.5% of the repositories have real vulnerabilities, and 37.5% do not have vulnerabilities. Hence, there are enough vulnerabilities available in open-source repositories to create a database of real security vulnerabilities. Furthermore, the most common security standards in those repositories are Injection (such as OS injection and SQLi), XSS, and Memory Leaks.

Zheng et al. [2021] created a vulnerability dataset based on the alerts reported by the SATs in real-world C/C++ programs (OpenSSL, FFmpeg, httpd, nginx, libtiff, and libav). The process consists of running the SAT *Infer* [Facebook, 2013] in a GitHub commit before and after a vulnerability fix. If the alert disappears from one version to the other, it is considered fixed. The commits were selected based on their commit message, and they used the NVD database [of Standards and Technology, 2020] to relate to a vulnerability report. Different from other existing datasets, which contain the vulnerable/non-vulnerable label at a function level, they define their samples from inter-procedural analysis. As the number

of samples was large and did not have a ground truth, they manually reviewed a few issues. With the resulting dataset, they used ML algorithms to reduce the reported false alarms. Among the analyzed metrics, they used the False Positive Reduction Rate, achieving results ranging from 65.5% to 95.8% for the six projects analyzed.

Fan et al. [2020] created the Big-Vul dataset with 3,754 vulnerabilities from 348 GitHub projects (*e.g.*, Chrome, Linux, Android, ImageMagick, Tcpdump, among others). These are collected from CVE Details, such as in our work. However, differently from [Fan et al., 2020], we enrich the vulnerability data with SMs and alerts from SATs. Additionally, our dataset has more vulnerabilities than Big-Vul.

2.3.4 ML for Software Security

Walden et al. [2014] use static data (extracted from the source code) as input (features) for several ML algorithms (*i.e.*, Decision Tree (DT), k-Nearest Neighbors (k-NN), Naive Bayes (NB), Random Forest (RF), and Support Vector Machine (SVM)) to detect software vulnerabilities. Two types of features (SMs and text mining) are extracted from PHP applications (Drupal, Moodle, and PHPMyAdmin). As the density of vulnerability is smaller than defects, defect prediction models cannot be used to predict vulnerabilities, and new ones have been suggested by the authors. They analyzed both recall and inspection ratio, and their results show that a higher recall is obtained when text mining is used when compared to software metrics. Overall, their results are better when using the text mining features in the three projects (*recall* varying from 0.737 to 0.805, while with SMs, it ranges from 0.663 to 0.769). This is probably related to the small number of SMs considered, which is 12. The results may also be overfitted for text mining, which leads to better performance.

Alves et al. [2016b] use different ML techniques in a software metric dataset to predict vulnerabilities in C/C++ projects. The dataset contains SMs of both before and after the patch of a vulnerability in the Common Vulnerability and Exposures (CVE), and it is created in a previous work of the authors [Alves et al., 2016a]. The SMs were extracted using the SciTools Understand tool [SciTools, 2011]. The analyzed projects are *Mozilla*, *httpd*, *glibc*, *Linux Kernel*, and *Xen Hypervisor*. The used ML classifiers are NB, DT, RF, and Logistic Regression (LR). Seven metrics were used to analyze the results: *Accuracy*, *FPR*, *Precision*, *Recall*, *F-Measure*, *Bookmaker Informedness*, and *Markedness*. The results varied a lot: *precision* varied from 0.32% to 30.50%, and *recall* from 0.36% to 100.00%.

Flynn et al. [2018] propose an approach to determine if the alerts reported by the SATs are actual vulnerabilities. They developed an alert aggregation tool from the multiple SATs and mapped them into the CERT Coding Rule [CERT, 1988] or CWE [MITRE, 2006a]. The used ML algorithms are Lasso Logistic Regression, Classification and Regression Trees (CART), RF, and Extreme Gradient Boosting (XGB). The validation technique used is the holdout method, in which the data are divided into training and test sets. Both per-rule classifiers and all-rules classifiers are created and evaluated. Some of the techniques used are alert type se-

lection, contextual information, data fusion, machine learning, and mathematical and statistical models. For their datasets, combined results of the SATs produce results with better accuracy than the tools separately. Results show poor accuracy in a per-rule prediction due to the little labeled data.

Pang et al. [2015] create an approach using N-Gram analysis and statistical feature selection to predict vulnerable software. *N-Gram* is the process of creating features by the composition of words in a text, with a varying number of words (N in this case). The text used to create features is the source code of the vulnerable code to be classified. For the experiment that the authors performed, four Android Java Applications were used. Additionally, statistical feature selection is used to reduce the number of features to be used in the prediction. Three metrics are analyzed (accuracy, precision, and recall), and the results are good when training and testing are done using data from the same project. However, the results are not so good when one project is used to train and another one to test.

Medeiros et al. [2017] studied whether SMs could be used to distinguish vulnerable and neutral (non-vulnerable) code at a file and function level and showed how a near-best subset of these metrics could be found. The dataset from Alves et al. [2016a] was used in this study, with the following projects: Mozilla Firefox, Linux Kernel, Xen Hypervisor, Apache httpd, and glibc. A heuristic search, which combined genetic algorithms with the RF classifier, was performed. The genetic algorithm converged to the best accuracy of 97.66% seven out of ten times at a file level. Complexity metrics were in most of the best solutions/models at the file level. In contrast, volume metrics (*e.g.*, Line of Code (LOC)) were the majority at the function level. Statistical analysis is done using Pearson and Spearman correlations, and the results are analyzed using the metrics accuracy and Cohen's Kappa. The best accuracy and Cohen's Kappa results were 97.30% and 37.03%, respectively.

Medeiros et al. [2020] use SMs to predict vulnerable code with ML algorithms (DT, RF, XGB, and SVM) at two levels of code units (file and function) of five C/C++ projects (Mozilla, Linux Kernel, Xen, httpd, and glibc) using the dataset by Alves et al. [2016a]. The algorithms with the best performance at the file level are RF and XGB in all the scenarios evaluated. The file-level results are usually better (recall about 0.900) than the function-level (recall about 0.800). Nevertheless, the prediction does not take into consideration the vulnerability type. Additionally, the work suffers from the problem of the use of SMs: the prediction indicates potentially vulnerable code without indicating the exact place of the vulnerability. Furthermore, the good results for recall hide a low precision, meaning that a large number of false alarms are raised. Consequently, additional time needs to be spent by development teams to identify the true vulnerabilities.

Li et al. [2018] proposes VulDeePecker, a deep learning-based approach to detect vulnerabilities that automatically selects features for the vulnerability detection process without human intervention. This is done using Code Gadgets, which are used as input for the deep learning algorithm. Code Gadgets are a vector representation of the functions. This is done with the help of the deep learning-based approach VulDeePecker. To evaluate VulDeePecker, they present a dataset of code gadgets with three projects (Xen, SeaMonkey, Libav), in which they were

able to discover four previously unknown vulnerabilities. The dataset has 61,638 code gadgets, 17,725 of which are vulnerable: 10,440 code gadgets with buffer overflow errors (*CWE-119* [MITRE, 2006b]), and 7,285 with resource management errors (*CWE-399* [MITRE, 2006e]). VulDeePecker could detect 4 vulnerabilities not previously reported in the NVD [of Standards and , NIST(2005)]. Nevertheless, these vulnerabilities were silently fixed in future versions of the evaluated software projects. Their results also show a lower FPR (5.7%) and False Negative Rate (FNR) (7.0%), when compared to the other techniques evaluated in the study. A key limitation of VulDeePecker is that it only deals with vulnerabilities related to library/API function calls (*e.g.*, `strcmp`).

Yamaguchi et al. [2014] develop the Code Property Graph (CPG) data structure to assist in the discovery of software vulnerabilities. The CPG is the merge of three other representations: Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). They use a graph database that contains the following vulnerability types: buffer overflows, integer overflow, format string vulnerabilities, or memory disclosures. They could detect 18 vulnerabilities previously unknown in the Linux Kernel using their approach. They also developed Joern [Yamaguchi, 2014], a tool used to extract the CPGs and the intermediate representations.

Bilgin et al. [2020] use a binary representation of ASTs to predict vulnerable C functions, using the Draper VDISC dataset [Russell et al., 2018], with functions of open-source projects such as Debian Linux. The dataset includes memory-related vulnerabilities, such as *CWE-119* (“Improper Restriction of Operations within the Bounds of a Memory Buffer” [MITRE, 2006b]) and *CWE-120* (“Classic Buffer Overflow” [MITRE, 2006c]). As the dataset is imbalanced, they used random undersampling, leading to a balanced dataset (50% vulnerable, and 50% non-vulnerable functions). Nevertheless, they ended up with a small dataset (no more than 2,684 vulnerable functions in the largest configuration). To decrease the processing time, they pruned the ASTs, and evaluated ASTs of different depths (of 6, 8, 10, and 12). Both a multi-layer perceptron (MLP) and Convolutional Neural Network (CNN) were used in their evaluations. For the best configuration, they had a precision of 70.1%, a recall of 52.1%, and an F-measure of 59.8%.

Wu et al. [2022] developed VulCNN, which is a method to detect vulnerabilities based on CNN. To evaluate their method, they used a dataset with 13,687 vulnerable functions and 26,970 non-vulnerable C/C++ functions from the Software Assurance Reference Dataset (SARD) from the National Institute of Standards and Technology (NIST), and the NVD. They extracted the program dependency graphs (PDGs) using Joern [Yamaguchi, 2014]. Their methodology uses an image-inspired approach. After extracting the PDGs, they use a sentence embedding technique (`sent2vec`) for each statement of the function. Then, they generate images for the functions by applying three centrality techniques (degree centrality, Katz centrality, and closeness centrality). Each one represents a channel of an “image” of the function. Finally, they train their classification model using a CNN model. They were able to detect 73 vulnerabilities previously unknown in a case study with three projects (Libav, Xen, Seamonkey).

Cao et al. [2021] propose a bidirectional graph to detect vulnerabilities using neural networks. To extract the information from the graphs, they combine the AST, the CFG, and the Data Flow Graph (DFG) into the Code Composite Graph (CCG). The CCG is the bidirectional graph that they use to extract the features with the Word2Vec algorithm [Mikolov et al., 2013]. To demonstrate their approach, they used 113 vulnerable functions (in 69 vulnerable files) of four projects (Kernel, FFmpeg, Wireshark, and Libav). Their approach can obtain a precision of 45.1%.

Wartschinski et al. [2022] create VUDENC, which is a Vulnerability Detection technique that uses Deep Learning. VUDENC automatically learns the features from Python codebase projects, and Word2Vec is used for that. Their dataset, which had 1,009 vulnerability-fix commits, included 7 types of vulnerabilities: SQLi, XSS, Command Injection, XSRF, Remote Code Execute, Path Disclosure, and Open Direct. Using a network of LSTM (long-short-term memory) cells, they could obtain a precision between 82.2% and 96.0% and a recall of 78.0% and 87.2%. The authors concluded that higher metrics (*e.g.*, precision, recall, and F-Measure) are obtained when using a single project and predicting if the whole file is vulnerable or not instead of predicting vulnerable functions.

Yan et al. [2019] present MAGIC, which is a technique to categorize malware programs using their CFGs as input. The approach consists of extracting features related to the vertex structure and code sequence for each node of the CFG. These are used as input to a Deep Graph Convolution Neural Network (DGCNN) and pooled using one of two techniques (Adaptive Max Pooling or Sort Pooling), before applying the network Visual Geometry Group (VGG). As part of the DGCNN process, they use the graph kernels defined by Zhang et al. [2018]. They could obtain results higher than 96% for both precision, recall, and F-measure.

2.3.5 Vulnerability Characterization and Prioritization

Chowdhury and Zulkernine [2010] conduct a statistical analysis to compare complexity, coupling and cohesion SMs with vulnerabilities. Their work aims to understand which level of metrics (design or code) are better indicators of vulnerabilities. The case study is conducted in several releases of the project Mozilla Firefox. The authors conclude that the metrics are positively related to the number of vulnerabilities, and code-level metrics are strongly correlated than design-level metrics.

The ISA approach (Integrated Software Source Code Analysis) is presented in [Kong et al., 2007]. The output of the different SATs (namely Rats, Its4, Flawfinder) are fused to take advantage of the different tools, and the outcome is presented in an XML format. Three C/C++ projects are analyzed (wu-ftp, net-tools, pure-ftp). Additionally, a score of the vulnerabilities is defined based on the severity reported by each tool for each vulnerability and a weight given to each SAT.

A similar approach is used by [Meng et al., 2008], in which the results of the SATs are merged based on the specification of the defect pattern. Moreover, two policies are used to prioritize the results, which use both the severity of the alert

and the count of each alert by multiple tools. The analyzed projects are written in Java, and the analyzed SATs are FindBugs, PMD, and JLint.

Neto and Vieira [2011] suggest an approach to use the output of different SATs to define a trustworthiness metric. Their focus is on SQLi vulnerabilities. Three SATs are used: SpotBugs (FindBugs at the time that the work was published), Yasca, and IntelliJ IDEA Analyzer. The metric normalizes the results to create a benchmark, allowing the comparison of different applications. Seven Java projects are used, and the metrics are analyzed.

Muske and Serebrenik [2016] conduct a survey that lists the main approaches to prioritize the alerts reported by the SATs. They are: *i) statistical analysis-based ranking* (either more simple approaches or based on Bayesian statistics), *ii) history-aware ranking* (the reasoning is usually based on the quickest fixes are usually the most important ones), and *iii) user feedback-based self-adaptive ranking* (using the user input to rank the alerts).

Le et al. [2022] surveyed the works that assess and prioritize software vulnerabilities using ML and deep learning. They categorized the key tasks and the attributes for performing such prioritization. Many studies focus on predicting the exploitation (exploit time, likelihood, or characteristics), impact, severity, type, and other miscellaneous characteristics. Other works focus on predicting the CWE or custom vulnerability types. A few other studies use other techniques to prioritize, such as rule-based methods, SCA, and socio-technical aspects.

Sharma et al. [2023] proposed VIEWSS (Variable Impact-Exploitability Weightable Scoring System), a scoring system for prioritizing software vulnerabilities. VIEWSS varies the weights of impact and exploitability to better understand the severity of the final score. The authors created a hybrid approach based on two other scoring systems (Vulnerability Rating and Scoring System (VRSS) [Liu and Zhang, 2011] and Weighted Impact Vulnerability Scoring System (WIVSS) [Spanos et al., 2013]) that use the six base metrics from CVSS [First, 1995]. Results show that VIEWSS provides scores to the vulnerabilities that have a close to normal distribution and can prioritize the more critical vulnerabilities better than the individual VRSS, WIVSS, and CVSS scoring systems.

Zeng et al. [2022] proposed LICALITY for prioritizing vulnerabilities. LICALITY uses a threat modeling method, neural networks, and probabilistic logic programming to extract information from vulnerability reports. To validate the approach, they used two datasets: (1) Microsoft Vulnerabilities [BeyondTrust, 2020] and (2) Advanced Persistent Threat Vulnerabilities [CISA, 2020]. Compared with CVSS and other prioritization approaches, results show that LICALITY can reduce the vulnerability remediation work of future threats by at least 1.85.

Medeiros et al. [2018] proposed a trustworthiness benchmark based on SMs. A score is calculated to indicate the trustworthiness of files or functions. A weight should be defined for each SM to calculate the trustworthiness score. Three techniques are used to define the weights: Mean Decrease Accuracy (MDA), Mean Decrease Gini (MDG), and MDAMDG (sum of MDA and MDG). In a validation phase, nine experts were asked to compare five files and five functions pairwise. Aggregated rankings of the files and the functions are obtained with the compar-

ison responses. Results indicate that the benchmark provides rankings similar to the experts. In another work, the same authors used SMs and ML to create trustworthiness models to characterize code units [Medeiros et al., 2023]. The models are based on the Simple Additive Weighting (SAW), which is also MCDM method. Each SM receives a weight from each ML model. That weight is used to calculate scores that are used to place code units into code categories representing the likelihood of being vulnerable or not. Results show that code units more prone to be vulnerable can be effectively distinguished.

2.4 Summary

This chapter presented the background concepts required to understand the studies and the solutions proposed in this thesis, including software security and vulnerability detection techniques. Also, related work was presented, including tools and techniques for vulnerability detection, vulnerability analysis, vulnerability datasets, vulnerability characterization and prioritization, and ML for security.

Despite the current focus on techniques for improving software security, vulnerabilities are still present in most software applications. Hence, developing new mechanisms to improve the characterization and detection of software vulnerabilities is essential. In the following chapters, we propose approaches to create a vulnerability dataset, detect software vulnerabilities, and characterize code units, always supported by the analysis of static data extracted from the source code.

Chapter 3

Building a Vulnerability Dataset

Developing techniques and tools to detect software vulnerabilities and characterize code units is not straightforward. It requires a large amount of data to validate such techniques. Although datasets with vulnerability information are available, they usually need to be updated or do not contain all the data required for the work. To support the development of the techniques later discussed, we created a dataset of software vulnerabilities collected from the CVE Details website [Özkan, 2023]. We enhanced with static information for both the vulnerable version of the source code and the fixed (neutral) one.

To build the dataset, we developed a generic and automated process capable of collecting software vulnerabilities from online vulnerability databases (in our case, CVE Details) and enriched them with data from bug trackers (*e.g.*, Bugzilla), version control systems (*e.g.*, GitHub), and static code information (SMs and alerts from SATs). In practice, for each vulnerability, we retrieve the corresponding source code units from the project repository (including both vulnerable and fixed versions) and then use SciTools Understand tool to compute a large set of SMs for those code units and run two SATs (Flawfinder and CppCheck) to collect security alerts (*i.e.*, potential vulnerabilities and/or weaknesses). The code units can be of three different levels: files, functions, or classes.

In short, the contributions of this chapter are two-fold:

- An automated process to collect the vulnerabilities from CVE Details and extract both SMs and SAT alerts from the vulnerable and non-vulnerable versions of the code collected from code repositories. This process can be used and/or adapted by other developers to build similar datasets for other projects.
- A dataset currently encompassing 5,214 vulnerabilities from five large open-source C/C++ projects: Mozilla, Linux Kernel, Xen, Apache httpd, and Glibc. The dataset is publicly available and can be easily updated with new vulnerabilities and new projects using a set of scripts that implement the proposed data collection process.

It is important to emphasize that our dataset can be used in many different contexts, including prediction of vulnerable code units, analysis of reported SAT alerts, identification of code not following best practices, etc. Each of the following chapters may use a different slice of this dataset, either referring to a specific period in time or a subset of projects. Thus, each chapter clearly details the slice being used.

The rest of this chapter is organized as follows. Section 3.1 presents the process developed for building the dataset. Section 3.2 presents and discusses the dataset itself. Threats to validity are discussed in Section 3.3. Finally, the chapter closes with a summary in Section 3.4.

3.1 Process to Build the Dataset

This section presents the automated process for building datasets of vulnerability metadata, including SAT alerts and SMs. The dataset is created from information initially collected from online sources and later analyzed by third-party tools. The steps are depicted in Figure 3.1. They are: (A) Retrieve the reported vulnerabilities from online platforms, such as CVE Details, security advisories from projects, as well as their bug trackers; (B) Retrieve vulnerable and neutral (patched) source code files from version control systems (*e.g.*, GitHub); (C) Generate SAT alerts and SM from the source code; and (D) Store the collected data in a database.

For convenience, we will refer to this diagram throughout each subsection. The term SAT is henceforth used to refer exclusively to tools that report possible vulnerabilities in source code, also known as SAT alerts. The scripts implementing the process detailed in the next sections can be accessed through the following URL: <https://eden.dei.uc.pt/~josep/thesis/>

3.1.1 Collect Vulnerabilities from Online Platforms

The process starts with a query to the CVE Details website (<https://www.cvedetails.com/>) to obtain a list of reported vulnerabilities for each selected project (“A. Retrieve Reported Vulnerabilities from Online Platforms” action on the top left corner of Figure 3.1). For each vulnerability, the information provided by CVE Details is analyzed, and any relevant fields in the metadata are saved, including: *i*) a unique identifier from CVE; *ii*) a value known as the CVSS score, which represents how severe it is given its properties and environment [Özkan, 2020]; *iii*) how much it impacts the confidentiality, integrity, and availability of a system; *iv*) how hard it is to exploit; *v*) whether or not authentication is required to exploit it; *vi*) zero or more vulnerability types; and *vii*) an optional numerical identifier known as CWE, which maps a vulnerability to a specific type of weakness.

The Common Weakness Enumeration (CWE) provides a catalog of known software and hardware weaknesses/vulnerabilities [MITRE, 2006a]. Including CWEs for all the vulnerabilities in the dataset allows grouping and studying the different vulnerability types. A challenge in the collection process is that, al-

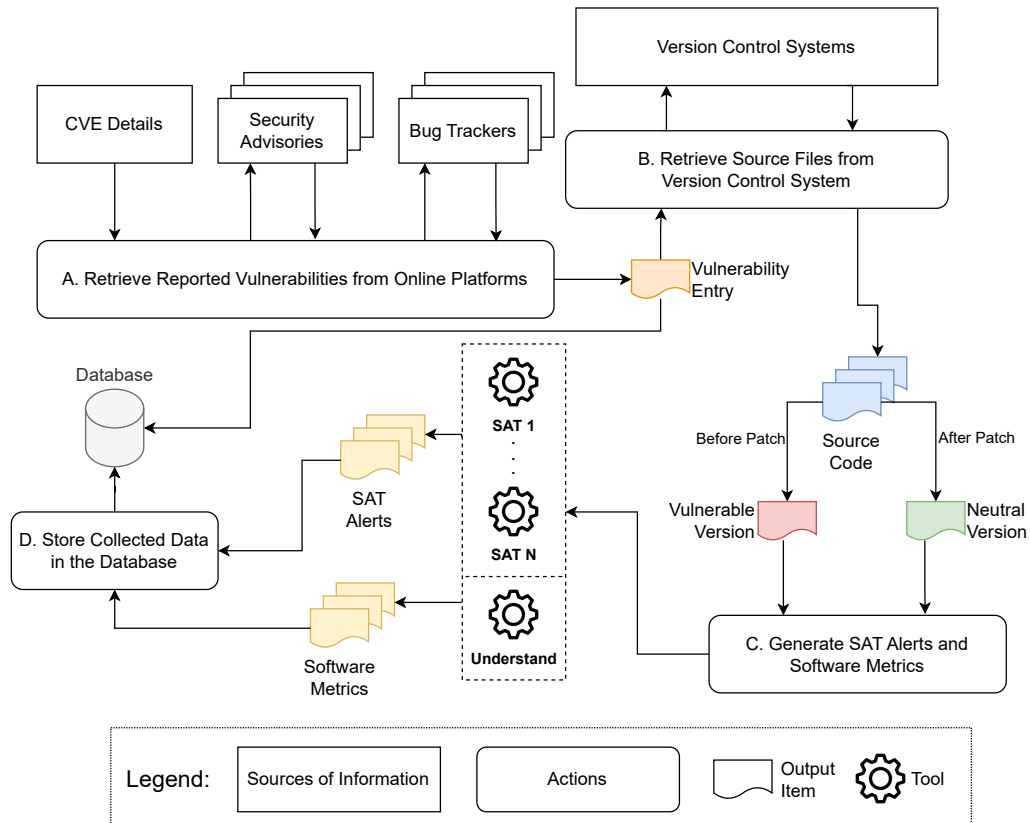


Figure 3.1: Dataset creation pipeline. This process begins with retrieving vulnerabilities’ metadata and ends with generating an enriched set of data samples for each code unit

though the CVE Details website has a field for the vulnerability types, it can have multiple values for each vulnerability. For instance, a vulnerability can have both the values *Denial of a Service* and *Execute Code*, which are consequences instead of the root cause of the vulnerability. For this reason, we decided to define a classification that allows us to reflect on the actual root cause of the vulnerabilities.

Several initiatives have already defined categories for vulnerabilities, such as the Seven Pernicious Kingdoms [Tsipenyuk et al., 2005]. However, they either focus on the vulnerability source or the consequence/effect in case it is exploited. For our dataset, we prefer categories that are actionable by the developer (e.g., in Chapter 7, we use these categories to characterize code units for the development team). Hence, we propose the use of a set of categories based on the groups of best practices defined by OWASP [Turpin, 2010]. This means that a vulnerability listed in a given category exists because the developers probably did not follow the OWASP guidelines in that same category. OWASP guidelines are complete and widely used, especially when studying about developing secure source code.

As shown in Table 3.1, most groups of best practices from OWASP are mapped into a category in our classification. The exception is the *Permission* category, which is composed of three OWASP categories: 1) Authentication and Password Management, 2) Session Management, and 3) Access Control. These three categories are related to the *Permission* aspects of the software system, and grouping them allows us to have a larger and more cohesive group than having them sep-

arate. In practice, CWEs can be mapped to a vulnerability category. For example, *CWE-119* [MITRE, 2006b] is assigned to the *Memory Management* category, and *CWE-20* [MITRE, 2006d] is assigned to the *Input Validation* category.

Table 3.1: The vulnerability categories considered for this work and examples of CWEs

OWASP Groups	Category	Examples of CWEs
Memory Management	Memory Management	119 (<i>Improper Restriction of Operations within the Bounds of a Memory Buffer</i>), 416 (<i>Use After Free</i>)
Input Validation	Input Validation	78 (<i>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</i>), 94 (<i>Improper Control of Generation of Code ('Code Injection')</i>)
Authentication and Password Management Session Management Access Control	Permission	284 (<i>Improper Access Control</i>), 287 (<i>Improper Authentication</i>)
Data Protection	Data Protection	199 (<i>Information Management Errors</i>), 200 (<i>Exposure of Sensitive Information to an Unauthorized Actor</i>)
Coding Practices	Coding Practices	19 (<i>Data Processing Errors</i>), 254 (<i>7PK - Security Features</i>)
Cryptography	Cryptography	310 (<i>Cryptographic Issues</i>), 261 (<i>Weak Encoding for Password</i>)
System Configuration	System Configuration	16 (<i>Configuration</i>), 260 (<i>Password in Configuration File</i>)
File Management	File Management	22 (<i>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</i>), 59 (<i>Improper Link Resolution Before File Access ('Link Following')</i>)
Output Encoding	Output Encoding	116 (<i>Improper Encoding or Escaping of Output</i>), 838 (<i>Inappropriate Encoding for Output Context</i>)
Error Handling and Logging	Error Handling and Logging	532 (<i>Insertion of Sensitive Information into Log File</i>), 209 (<i>Generation of Error Message Containing Sensitive Information</i>)
Communication Security	Communication Security	319 (<i>Cleartext Transmission of Sensitive Information</i>), 419 (<i>Unprotected Primary Channel</i>)
Database Security	Database Security	202 (<i>Exposure of Sensitive Information Through Data Queries</i>), 502 (<i>Deserialization of Untrusted Data</i>)

Each CVE page lists several relevant external references, including the software changelog, discussion boards, the project bug tracker, and the version control system. These references may contain several hyperlinks, some of which link to the version in the code repository where the vulnerability was patched. Specifically, we retrieved an identifier known as *commit hash* that allowed us to locate any files affected by the vulnerability in a given version (*i.e.*, the commit) of the software (these files are later retrieved and processed). As this identifier is not available in some cases, we may need to consult other references. In practice, missing references in CVE Details can be handled by searching two other sources: the project bug tracker (*e.g.*, Mozilla uses Bugzilla to store the issues containing references to the commit fixes) and the security advisories (*e.g.*, Mozilla uses the MFSA - Mozilla Foundation Security Advisories). Both contain specific identifiers or keywords that may, in turn, be used to find the commit associated with a given CVE in the code repository. In the cases where no further information can be obtained (about one-third of the reported vulnerabilities in our case), the vulnerability cannot be considered for the upcoming steps of the process. Without references for the commits, we cannot identify the source code changes that fix the vulnerabilities (*i.e.*, we could not obtain the vulnerable code and the neutral code).

3.1.2 Retrieve Source Files from Version Control Systems

We used the commit hashes (obtained when gathering the metadata of the vulnerabilities) to collect the vulnerable version (file(s) before the vulnerability commit changes) and the neutral version (file(s) after the vulnerability commit changes) for each vulnerability (“B. Retrieve Source Files from Version Control System” action on the top right corner of Figure 3.1).

This step is performed by interfacing with the version control system of the project and requesting the files for a specific version. Although this method applies to different version control systems, in this work, we focused on Git repositories (because the selected projects are available on this version control system). The Git `rev-list`¹ command can be used to list every file path affected by a commit, as well as each commit’s tag name, author date, and a set of line number ranges that show where each file was modified. In practice, we followed the approach used in other works [Alves et al., 2016a; Fan et al., 2020; Zheng et al., 2021], and obtained the (neutral) code version where the vulnerability was corrected (patched) and the one immediately before the patch (vulnerable).

Any file present in a neutral commit or a commit not affected by a vulnerability is assumed to be neutral. This is represented in Figure 3.2, where all the files modified by a commit are considered vulnerable in the previous commits and the remaining ones as neutral. Likewise, the smaller granularity code units present in the file (*i.e.*, classes, functions) are also seen as non-vulnerable. For a vulnerable commit, each affected file is assumed to be vulnerable, but the same might not be true for its code units. For example, a vulnerable file with five functions may have only been patched in one of them. Hence, we need extra care in checking whether the changed lines overlap with any code units inside vulnerable files. Consequently, the code units labeled as vulnerable are only the ones changed in the vulnerability commit fix. The Clang compiler² can be used to locate functions and classes inside files. Note that because structs and unions from C source files are categorized as classes by the tool used to extract the Software Metrics (SciTools Understand), we decided to follow that same approach.

3.1.3 Generate SAT Alerts and Software Metrics

In this step, third-party tools are used to perform SCA in the C/C++ code units (files, functions, classes), and extract security alerts (SAT alerts) and SMs (“C. Generate SAT Alerts and Software Metrics” action in the middle right side of Figure 3.1) for the files identified during the execution of the previous step.

To generate security alerts many SATs can be used, including both commercial and open source (*e.g.*, Parasoft CppTest, SonarQube, Clang Static Analyzer, Framac, Facebook Infer, Oclint, Sparse, Splint, and Uno). Due to licensing restrictions and considering their wide use, and due to restrictions mentioned above, in this work, we resort to the following two open-source tools:

¹<https://git-scm.com/docs/git-rev-list>

²<https://clang.llvm.org/>

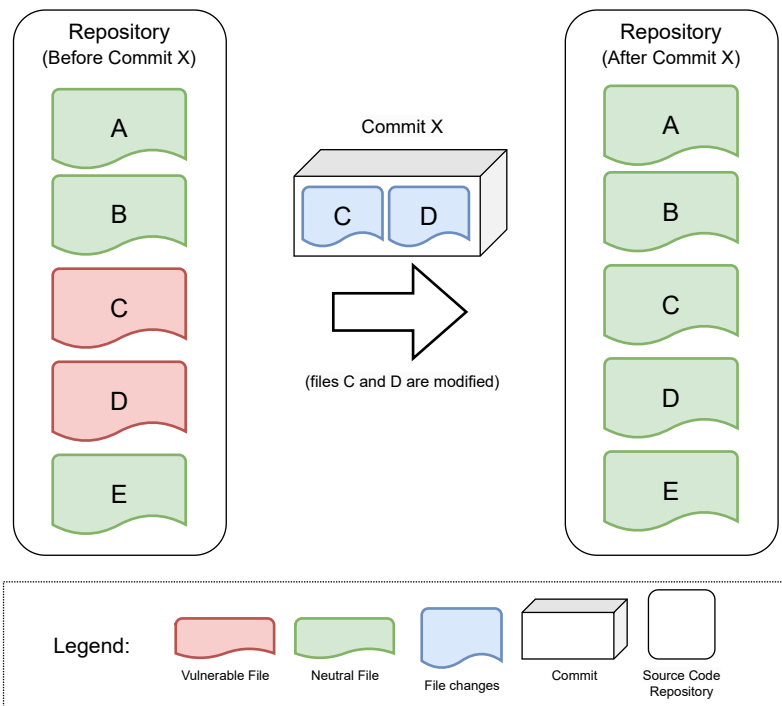


Figure 3.2: Source code repository representation before and after a commit that fixes a vulnerability

- (A) **Flawfinder** is an open-source SAT for C/C++ code [Wheeler, 2001]. It uses simple text pattern matching, and it ignores comments and strings. Flawfinder provides the following data for each alert: filename, line, level of severity, category, name and message of the alarm, and the CWE. Flawfinder is considered CWE-compatible. Flawfinder version 2.0.10 was used in this study (currently, it is in version 2.0.19).
- (B) **CppCheck** is another open-source SAT for C/C++ code [Marjamäki, 2007]. CppCheck provides the following data for each alert: filename, line, severity, alert identifier (with a message), and CWE. A large number of security issues have been detected in many projects with CppCheck, which can be integrated into a high number of development tools. In this study, we used version 1.82 (it is currently in version 2.12).

Although there are many open-source C/C++ SATs available, many of them use a build automation tool make. Many projects use make to build the software, but not all of them. For example, Mozilla was built using make for C/C++ code, but mach³ started being used in 2012, as it is tailored for the Mozilla needs. On the one hand, it eases the build process for the developers; on the other hand, it hinders the use of some SATs that require make to run. mach is based on make, but the latter cannot be directly invoked since the introduction of the former. The current version of mach allows performing static analysis on the source code (based on CLang). However, it was introduced only after 2016, and the dataset has vulnerabilities reported in versions from 2000 onward. Therefore, no SAT

³https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/mach

based on `make` can be used to identify the alerts in all cases. Otherwise, we end up with alerts only for part of the vulnerable code versions but not for all of them, as `mach` restricts the use of `make`.

The output of the SATs are alerts about potential weaknesses or vulnerabilities located on a specific line of a source file. These tools take the source files as input and output their analysis in a textual file format, usually a CSV (comma-separated values) file. As we mapped the initial and final line of code corresponding to each function and class, we could assign each SAT alert to the corresponding code unit. Flawfinder and CppCheck were selected as they are widely used and do not require compiling and building the source code. Consequently, it takes less time to run in the complete project repository. Nevertheless, our process is generic and can be applied using other SATs. Examples of an alert type reported by these tools are *strcpy* (Flawfinder issues an alert of this type when the code is using the unsafe `strcpy` function) and *nullPointerRedundantCheck* (CppCheck issues this type of alert when the code checks more than once if a variable is null). Flawfinder provides 126 rules for different alert types, while CppCheck provides 228 rules. Examples of rules for Flawfinder and CppCheck can be seen in Tables 3.2 and 3.3, respectively.

Table 3.2: Examples of SAT rules from Flawfinder. Each example corresponds to the use of a vulnerable function

Rule Category	Example of Rule
buffer	strcpy
misc	fopen
race	chown
shell	ShellExecute
tmpfile	GetTempFileName

Table 3.3: Examples of SAT rules from CppCheck (complete list can be seen online: <https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/>)

Rule Category	Example of Rule
Check function usage	memset() third argument is zero
Exception Safety	Throwing exceptions in destructors
Leaks (auto variables)	Detect when a auto variable is allocated but not deallocated or deallocated twice.
Memory leaks (address not taken)	Not taking the address to allocated memory
Null pointer	null pointer dereferencing

The next step is to calculate the SMs for both the vulnerable and neutral versions of the source code. Although there are many tools to support this step, we used SciTools Understand⁴ version 4.0.837. This version was also previously used to collect software metrics for vulnerable code by Alves et al. [2016a]. Furthermore, Understand is widely used in the field in projects with different programming languages and has also been considered in other research works, such as [Shin et al., 2011] and [Sultana et al., 2021]. The complete list of 54 metrics (related to

⁴ <https://www.scitools.com/>

complexity, volume, cohesion and coupling, etc.) can be seen in Appendix B and can also be obtained online⁵.

3.1.4 Store the Data in a Database

After collecting the vulnerability metadata and generating the SAT alerts and SMs for the vulnerable and neutral (non-vulnerable) versions, the data should be stored in a database to simplify the analysis and the creation of relationships among different entities (the action of storing the collected vulnerabilities, SAT alerts, and software metric information in a database is represented by the “D. Store Collected Data in the Database” step in Figure 3.1).

To accomplish this, we redesigned and improved the relational model proposed by Alves et al. [2016a], where entities such as vulnerabilities, code units, and patch information are related to one another based on their attributes. In practice, we inherited the base design and augmented it with additional fields and tables to store other types of data, namely the security alerts, SAT properties, vulnerability categories, and CWEs. With these new additions to the database schema, we can easily export datasets focusing on selected (or all) vulnerabilities and including information about the corresponding code units, namely Software Metrics (SMs), SAT alert occurrences, vulnerability categories, and whether or not each code unit was affected by a CVE. The data can be easily extracted and tailored according to the needs of the specific research at hand by means of simple SQL queries that slice and dice the data.

3.2 Dataset Characterization

Table 3.4 overviews the five open source C/C++ projects included in our dataset: Mozilla (*i.e.*, the source code for the Firefox browser), the Linux Kernel, the Xen Hypervisor, the Apache HTTP Server (httpd), and the GNU C Library (Glibc). In practice, the dataset includes projects with different sizes, complexities, and security needs. For example, Mozilla and Linux Kernel are the largest projects, with 24.6 million and 21.6 million lines of code, respectively. On the other hand, the other three projects (xen, httpd, and glibc) are much smaller, with less than 10% the size of the two largest projects. Projects of similar sizes have a number of known vulnerabilities in the same order of magnitude.

A summary of the number of vulnerabilities, patches, and vulnerable code units

⁵ <https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have->

⁵ <https://www.openhub.net/>

⁶ <https://github.com/mozilla/gecko-dev>

⁷ <https://github.com/torvalds/linux>

⁸ <https://xenbits.xen.org/gitweb/?p=xen.git;a=summary>

⁹ <https://svn.apache.org/repos/asf/httpd/trunk/>

¹⁰ <https://github.com/apache/httpd>

¹¹ <https://sourceware.org/git/glibc.git>

Table 3.4: A summary of the five large C/C++ projects used in our work. The total number of lines of code was taken from the Open Hub website⁶ on January 2022

Project	Language	Version Control	Lines of Code	Website
Mozilla	C++	Git ⁷	24.6 million	Mozilla.org
Linux Kernel	C	Git ⁸	21.6 million	Kernel.org
Xen Hypervisor	C	Git ⁹	0.6 million	XenProject.org
Apache HTTP Server	C	SVN ¹⁰ and Git ¹¹	1.5 million	Apache.org
GNU C Library (Glibc)	C	Git ¹²	1.4 million	Gnu.org

Table 3.5: A summary of the number of patches and vulnerable code units considered for the generated datasets

Project	Vuln.	Patches	Files		Functions		Classes	
			Vuln.	Total	Vuln.	Total	Vuln.	Total
Mozilla	2,215	5,917	12,418	1,787,723	8,643	48,185,503	2,302	5,043,307
Linux	2,355	1,523	43,639	1,971,130	9,004	32,793,147	120	4,403,349
Xen	306	649	878	25,982	965	221,375	21	22,571
Httpd	230	115	201	8,632	154	60,433	2	3,647
Glibc	108	96	322	141,447	105	215,720	3	23,630
Total	5,214	8,300	57,458	3,934,914	18,871	81,476,178	2,448	9,496,504

for each project is also included in Table 3.5. A patch is a commit for a vulnerability, which may take more than one commit to be fixed. Note that this table shows more vulnerabilities than patches in some of the projects since it is not always possible to associate a commit hash to every CVE given the information available in CVE Details, bug trackers (e.g., Mozilla project uses Bugzilla: <https://bugzilla.mozilla.org/>), and security advisories. A possible reason is that the vulnerabilities may be in third-party components used by these projects and not in their own source code. Consequently, these third-party components need to be updated without modifying the project’s source code.

Mozilla and Linux Kernel account for most of the vulnerabilities. The latter has a number of vulnerabilities (2,355) larger than the former (2,215), even if Linux has a smaller codebase. This probably happens because Linux is a project older than Mozilla. Among the smallest projects, we can highlight that Xen has many more reported vulnerabilities than Apache httpd and glibc, even if it has the smaller code base of the five projects.

To better understand the dataset, let us analyze the vulnerabilities included as well as their categories. As mentioned before, we mapped the vulnerabilities to categories based on their CWE. In practice, we mapped the CWEs to the corresponding categories, which are based on OWASP best security coding practices [van der Stock et al., 2017] (e.g., *CWE-416 (Use After Free)* [MITRE, 2006f] was assigned to *Memory Management* category). The resulting distribution per category can be seen in Table 3.6. The categories with the largest number of reported vulnerabilities for the selected projects are *Memory Management* (27.57%), *Input Validation* (14.77%), and *Permission* (10.15%).

Table 3.6: The vulnerability categories and the respective number of vulnerabilities.

Category	CWEs	Vulnerability Count (%)
Memory Management	119, 362, 399, 416, 476, 824	1437 (27.57%)
Input Validation	20, 78, 79, 91, 94, 134, 189	770 (14.77%)
Permission	255, 264, 269, 284, 287, 352	529 (10.15%)
Data Protection	199, 200	435 (8.34%)
Coding Practices	17, 19, 254	92 (1.76%)
Cryptography	310	44 (0.84%)
System Configuration	16	23 (0.44%)
File Management	22, 59	23 (0.44%)
Output Encoding	-	0 (0.00%)
Error Handling and Logging	-	0 (0.00%)
Communication Security	-	0 (0.00%)
Database Security	-	0 (0.00%)
Missing CWEs	-	1861 (35.69%)
Total	-	5214 (100%)

A detailed analysis of the dataset shows that the categories with the largest number of reported vulnerabilities remained the same over the years. However, the most frequent CWEs have been evolving. This can be observed in Figure 3.3. The chart shows the evolution of the number of reported vulnerabilities with the same CWE. Up until 2012, the most frequent was *CWE-399 (Resource Management Errors)* [MITRE, 2006e]. From 2013 to 2019, *CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer)* [MITRE, 2006b] was the most reported CWE in 4 out of the 7 years, being in the top 3 in all of them. Both *CWE-399* and *CWE-199* belong to the *Memory Management* category, which accounts for about one-third of the reported vulnerabilities in the dataset. As shown in Table 3.6, the remaining most frequent vulnerability types belong to the following categories: *Input Validation*, *Permission*, and *Data Protection*.

The dataset has been built using the process and scripts described in the previous section. Overall, the process of collecting the vulnerabilities from CVE Details and their patches took about four days to execute for the five projects. The process of locating every file affected by a vulnerability and extracting its functions and classes took roughly three weeks. Following that, generating software metrics for all projects took about two weeks. The complete database is available at the following URL: <https://eden.dei.uc.pt/~josep/thesis/>

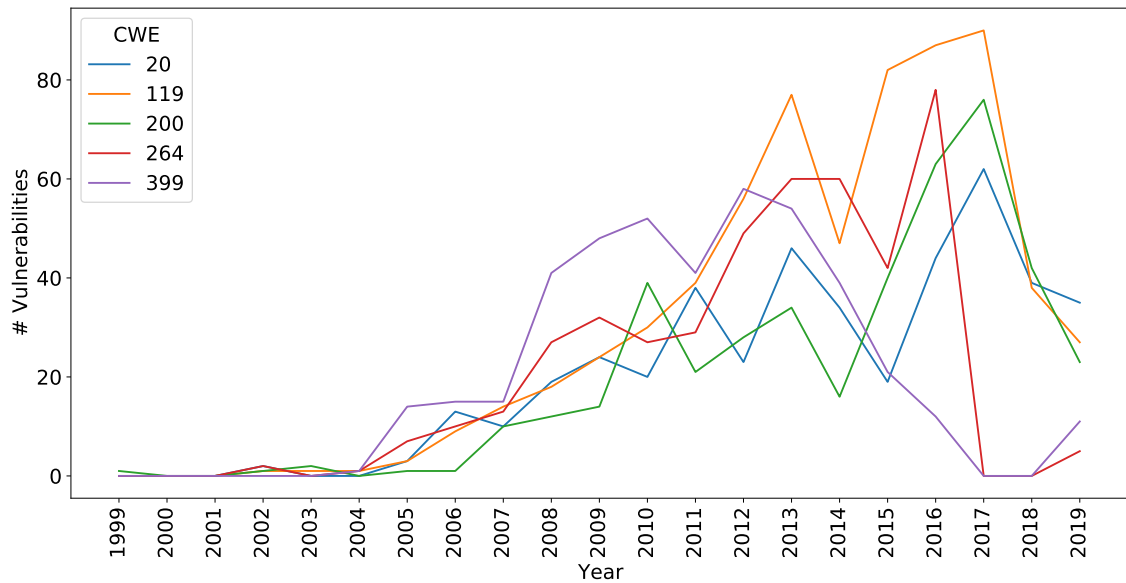


Figure 3.3: Analysis of the most common vulnerability types (CWEs) along the years

3.3 Threats to Validity

This section discusses weaknesses that might have influenced the construction of the dataset.

External Validity refers to the ability to generalize the results. Five C/C++ open-source projects were considered for the vulnerabilities included in the dataset. Although the number of projects considered is not big, they are widely used. Moreover, all the projects are coded in C/C++ programming languages, the number of vulnerabilities per category is not large, and the information collected for each vulnerability is restricted to SMs and SAT alerts. Nevertheless, the dataset is prepared to be updated with more vulnerabilities of the selected projects as well as with other projects (and generic scripts are provided for doing so). Also, other SMs and alerts from more SATs can be easily added.

Internal Validity refers to the possibility of having unanticipated relationships. Due to the output format of SciTools Understand and Clang, possibly some SAT alerts were not adequately assigned to functions. SciTools Understand does not emit line numbers, meaning that many code units had to be associated by name. Moreover, it was noted that Clang does not always include certain functions or classes in the AST output. Put together, this means that while a SAT may have generated alerts for a function, the corresponding entry in the database might be missing its line numbers (which happens for about 15% of the functions). Thus, vulnerable functions can become associated with zero alerts, although they exist and could represent potential vulnerabilities.

Construct Validity refers to the vulnerability categories that we have proposed, and mistakes may have happened in the mapping of CWEs into categories. Although we followed the OWASP best practices as guidelines and carefully re-

viewed the CWEs, some CWEs may have been misplaced. Consequently, the results per category could slightly differ if a different assignment was done.

3.4 Summary

In this chapter, we presented an automated process for building datasets of vulnerable code units along with data collected using SCA. We showed how the online platform CVE Details can be scraped for vulnerability metadata, how we can find any affected files, functions, and classes by querying the version control system of the project, and how these distinct kinds of data were collated into a relational database. We enriched this data with Software Metrics using the tool SciTools Understand and SAT alerts using the SATs Flawfinder and CppCheck.

The dataset is composed of vulnerabilities from five open-source C/C++ projects (Mozilla, Linux Kernel, Xen, Apache httpd, and Glibc). The categories with more vulnerabilities are *Memory Management*, *Input Validation*, *Permission*, and *Data Protection*. These indicate the main areas that software developers of C/C++ should pay more attention to avoid security issues. Although the projects used to create the dataset are all of the same programming language, the proposed methodology is generic and can be replicated in other projects using different programming languages. It is important to emphasize that the content of the dataset evolved throughout the development of the work presented in the thesis (to include vulnerabilities reported over the years).

The following chapters are based on the dataset presented in this chapter. However, not the entirety dataset is used in all experiments (for practical reasons and due to the evolution of the dataset over the time). In each chapter, we detail the dataset slice being used. We refer to the term *dataset slice* to indicate the portion of data, and each *dataset slice* can be a restriction of a project(s), a timeframe, or both. For instance, the work in the following chapter uses only the vulnerabilities of the Mozilla project reported from 2000 to 2016. In this case, the *dataset slice* refers to the project and timeframe.

Chapter 4

Evaluating Static Analysis Tools in Large Projects

Static Code Analysis (SCA) can be used from the early stages of the development life cycle, and its execution can be automated. In fact, a large set of Static Analysis Tools (SATs) are available nowadays, following different approaches to report potential problems (which we call alerts). The problem is that most tools report a high number of FPs and miss actual vulnerabilities (TPs), leading developers to disregard their use [Imtiaz et al., 2019]. This issue is even more critical when the project has a large code base and the use of SATs was not considered from the beginning of the software development. Hence, the software development team must handle a large technical debt when it starts using it.

Vulnerabilities have different characteristics, and different SATs can be more suitable to some of them. This chapter aims to understand the applicability of SATs in a large project and the vulnerability categories they can detect. Thus, we established the following Research Question (RQ):

- **RQ1:** Are SATs effective in detecting different types of software vulnerabilities in large-scale software projects?

In practice, we **run two SATs in a well-known and widely used and large software system written in C/C++** to collect performance metrics (precision, recall, f-measure) that allow comparing alternative tools. Results show that CppCheck could detect 83.5% of the vulnerabilities, and Flawfinder could detect 36.2%, although the number of false alarms is high (7.2% for CppCheck and 93.2% for Flawfinder). Regarding the different categories, the two SATs showed quite diverse performances (e.g., CppCheck could detect 92.6% of *Data Protection* vulnerabilities and 62.5% of *Coding Practices* vulnerabilities, while false alarms are 99.1% and 99.9%, respectively). In general, we can confirm the performance challenges faced by SCA tools.

The remainder of this chapter is organized as follows: Section 4.1 discusses additional details on the vulnerabilities and presents the approach followed in the

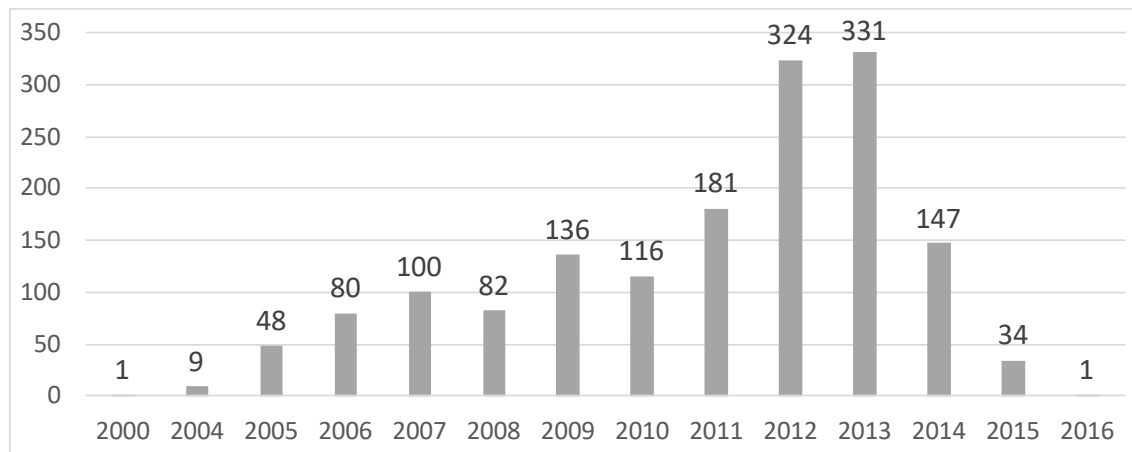


Figure 4.1: Number of commits that fixed vulnerabilities

study. Results are presented and discussed in Section 4.2. Section 4.3 discusses threats to validity. Section 4.4 closes this chapter with a summary.

4.1 Vulnerabilities and Approach

The *dataset slice* used to support our study consists of the vulnerabilities discovered between 2000 and 2016 in the Mozilla project¹. Mozilla is one of the projects with the most significant number of vulnerabilities in the dataset (a total of 2,441 at the time of the analysis in 2020). Such vulnerabilities are fixed in 1,590 patches/commits, and 1,251 files were changed by the vulnerability fixes. The vulnerabilities are of different categories, being the most frequent ones: improper input handling (13.03%) and incorrect memory management (24.42%). Our results show a wide range of performance values (precision from 0.00 to 0.71 and recall from 0.19 to 0.93) for the different vulnerability categories.

Figure 4.1 shows the number of Mozilla commits (known as *patches*) that fixed vulnerabilities from the dataset (a total of 1,590). The number of files in the repository is significant, and it varied from 15,900 to 48,612 files during this period. Also, the interval between the first two patches is of four years, as the first is of 2000 and the second is of 2004. Most of these vulnerabilities (more than two-thirds) were fixed after 2010, with more fixes in 2012 and 2013.

A summary of the number of vulnerabilities per category is presented in Table 4.1. Not all CWEs are presented in the table, but only the CWEs for the Mozilla project slice that we are considering for this study are presented. We observe that almost half (49.61%) of the vulnerabilities do not contain an associated CWE. They are either listed as “CWE id is not defined for this vulnerability” on the CVE Details website (980 vulnerabilities) or do not have data on CVE Details (2 vulnerabilities: CVE-2012-3977 and CVE-2014-1495). Moreover, 229 vulnerabilities are listed only on the Mozilla bug tracker but do not have a record in CVE Details. Hence, we do not consider such vulnerabilities when analyzing the SATs performance

¹ At the time of this particular work, the dataset included only vulnerabilities until that year.

Table 4.1: Vulnerability categories for the Mozilla dataset

Category	CWEs	Count (%)
Memory Management	CWE-119, CWE-399, CWE-362	596 (24.42%)
Input Validation	CWE-20, CWE-79, CWE-189, CWE-94	318 (13.03%)
Permission ²	CWE-264, CWE-255, CWE-284, CWE-287, CWE-352	177 (7.25%)
Data Protection	CWE-199, CWE-200	65 (2.66%)
Coding Practices	CWE-17, CWE-254, CWE-19	31 (1.27%)
Cryptography	CWE-310	18 (0.74%)
System Configuration	CWE-16	16 (0.66%)
File Management	CWE-22, CWE-59	9 (0.37%)
Output Encoding	-	0 (0.00%)
Error Handling and Logging	-	0 (0.00%)
Communication Security	-	0 (0.00%)
Database Security	-	0 (0.00%)
CWEs not found	-	1,211 (49.61%)
Total	-	2,441 (100.00%)

per category of vulnerability (but we consider them in the analysis of the overall performance of the tools).

The **approach followed in this study** is represented in Figure 4.2. The vulnerable and neutral (non-vulnerable) snapshots of the source code are analyzed using the SATs. As the dataset provides information about the files changed to fix each vulnerability, we used this information to run the SATs, following two approaches:

- i) only the vulnerable files are submitted to the SAT (both vulnerable and fixed versions, *i.e.*, commits before and after fix)
- ii) the complete repository (again, vulnerable and fixed/neutral versions) is submitted to the SAT

Although the second approach is more complete, it takes more time to obtain the alerts. Hence, we use both approaches in a complementary way. While approach *i*) shows if a SAT can detect the known vulnerabilities (*i.e.*, the ones in the dataset), approach *ii*) allows us to obtain vulnerability alerts for the entire source.

Using the alerts, we evaluate the SATs both on their ability to detect all vulnerabilities (regardless of their category) and per category. As the number of reported alerts is too large to be analyzed manually, we compared the number of alerts

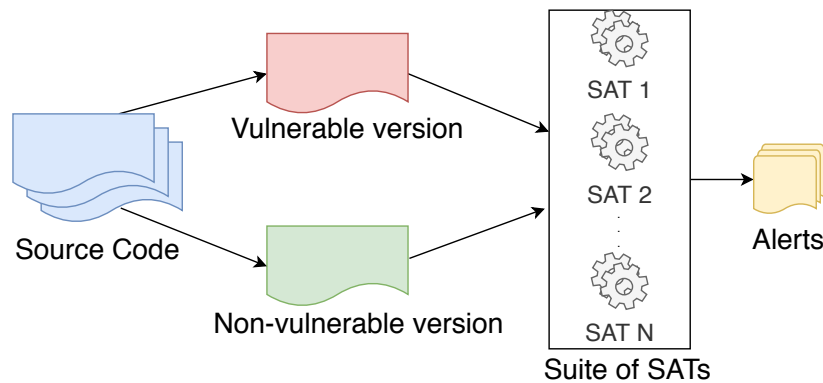


Figure 4.2: Methodology to run the SATs in the different snapshots source code

reported in the vulnerable and the neutral (non-vulnerable) versions. If the number of alerts decreases in the neutral version, we conclude that the SAT can detect that vulnerability.

The generic approach described above involves executing the SATs to obtain the alerts. However, as we already have those alerts in the dataset described in Chapter 3, we resorted directly to it in order to facilitate the experiments.

4.2 Results and Discussion

This section presents and discusses the results obtained. We start by analyzing the overall results in terms of the vulnerabilities detected by the SATs (also considering the evolution in the number of alerts), and then we present the global performance metrics for each tool. Finally, we discuss the performance of the SATs per vulnerability category.

4.2.1 Overall Observations

The first analysis is related to the number of vulnerabilities identified by the SATs. We use the alerts for the source code files of the 1,590 patches (commits) related to the vulnerabilities considered. Note that the number of patches (1,590) is smaller than the number of vulnerabilities (2,441). This happens for two main reasons: *i*) some patches fix more than one vulnerability, and *ii*) some vulnerabilities are fixed without code changes in C/C++ files (*e.g.*, integration with libraries). This was done both for the commit fixing the vulnerability and for the previous one (approach *i*) as described in the previous section).

Figure 4.3 shows a Venn diagram with the relation between the known vulnerabilities and the ones reported by the two tools. In this case, the total number of patches/commits is 1,017, as 573 out of the 1,590 patches/commits are not related to vulnerable files included in the dataset (*i.e.*, refer to non-C/C++ files

¹Permission is composed of *Authentication and Password Management*, *Session Management*, and *Access Control*.

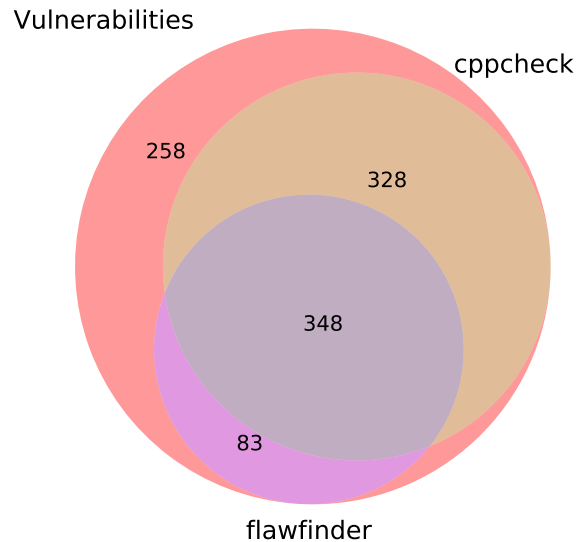


Figure 4.3: Number of patches/commits detected by the SATs

such as header files, which are not the target of the analysis). As shown, only 348 (34.22%) of the patches are reported as vulnerable by both SATs. 328 vulnerabilities (32.25%) are reported only by CppCheck, while a smaller number is reported only by Flawfinder (83 vulnerabilities, 8.16%). One-fourth of the known vulnerabilities (258, 25.37%) are not identified by any of the SATs. This shows that the use of these SATs would leave a high number of vulnerabilities undiscovered if SCA is the only technique used to detect them.

As it is not feasible to analyze all the alerts manually to confirm the reports from the SATs, we decided to use the following approach: if the number of alerts in a file changes from the vulnerable version to the fixed (neutral) version, we consider that the SAT can detect the original file as vulnerable; otherwise, the SAT cannot detect the vulnerability. Although this is a weak approach, it is adequate to have a first idea of how precise the SATs are.

To better understand the results, we continue with an analysis at the file level. Considering all the known vulnerable files over time, the goal is to understand which ones are reported by the SATs. The corresponding Venn diagram is in Figure 4.4. Note that, as before, we consider only the files with C/C++ extensions as the vulnerabilities are in these files.

From all the vulnerable files (1,251), the majority is reported by either CppCheck or Flawfinder (86.01%). However, the two SATs report a high number of potential FPs. CppCheck reports 661 FPs files, representing 38.74% of the files reported. Regarding Flawfinder, the number of FPs files is even more significant since 8,590 files (94.60%) do not contain actual known vulnerabilities. Although we consider these as FPs, some may indeed refer to unknown vulnerabilities (we cannot manually validate them due to the very large number of alerts). In any case, as the number of potential FPs is so high, even if some are actual vulnerabilities, our conclusions do not change.

Figures 4.5 and 4.6 show that, in general, the number of alerts increases for both CppCheck and Flawfinder over time (following approach *ii*) described in the pre-

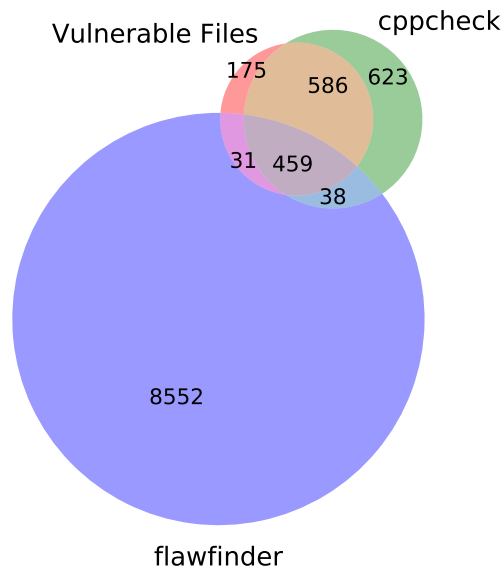


Figure 4.4: Vulnerable files reported by the SATs with all C/C++ extensions in the relevant directories

vious section). Both SATs have an increase in the number of alerts for patches from 2000 to 2016. This is an expected behavior since the source code increased as more functionalities were added to Mozilla. Also, CppCheck reports more alerts over time (from about 20,000 to 45,000) than Flawfinder (5,000 to 22,500). As we can see in Figure 4.4, the number of distinct files reported by Flawfinder is much more significant than the ones reported by CppCheck. As a consequence, we can conclude that Flawfinder alerts are more spread in multiple files and that CppCheck alerts are more concentrated in some files.

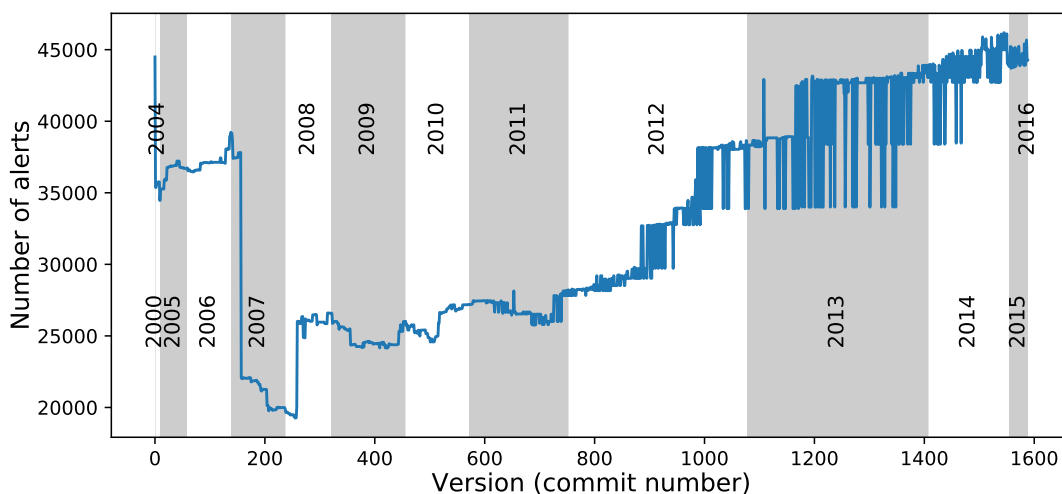


Figure 4.5: Number of alerts per commit along the time (CppCheck)

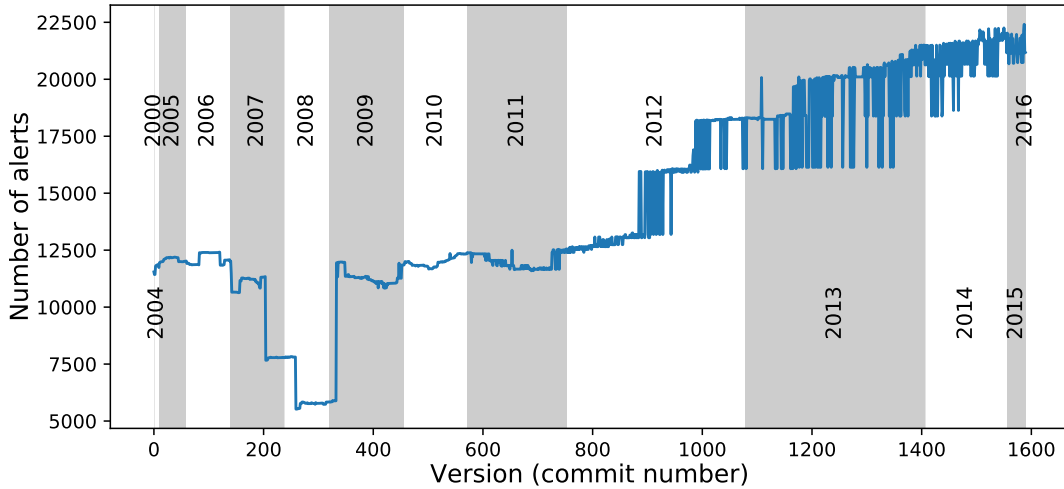


Figure 4.6: Number of alerts per commit along the time (Flawfinder)

4.2.2 Performance of the Tools

Each alert raised by a SAT falls in one of the following four cases: *i) True Positive (TP)*: an actual vulnerable file reported by the SAT; *ii) FN*: an actual vulnerable file not reported by the SAT; *iii) False Positive (FP)*: a file reported by the SAT as vulnerable that is not vulnerable; and *iv) True Negative (TN)*: a non-vulnerable file that is not reported by the SAT. This is the basis for computing the performance metrics of interest (precision, recall, and f-measure). Table 4.2 presents the classification results and the performance metrics for each SAT.

Table 4.2: SAT Results and Metrics - (A): CppCheck, (B): Flawfinder

	TN	FP	FN	TP	Precision	Recall	F-measure
(A)	92.8% (8,552)	7.2% (661)	16.5% (206)	83.5% (1,045)	0.61	0.84	0.71
(B)	6.8% (623)	93.2% (8,590)	60.8% (761)	36.2% (490)	0.05	0.39	0.10

As noticed in the previous section on the analysis of the Venn diagrams, CppCheck presents better results than Flawfinder. In addition to the higher number of FPs reported by Flawfinder (8,590), the number of TPs is significantly lower (490) than the one reported by CppCheck (1,045). Consequently, Flawfinder has worse values than CppCheck for the three performance metrics. Although one of the SATs clearly performs better (CppCheck) than the other (Flawfinder), the overall results are not satisfactory. Thus, in the following section, we analyze their performance when considering individual vulnerability categories.

4.2.3 Performance per Vulnerability Category

The same analysis can be done per vulnerability category. The detection results for CppCheck and Flawfinder can be seen in Tables 4.3 and 4.4, respectively, and the performance metrics in Tables 4.5. Regarding the categories with more known

vulnerabilities (*Memory Management* and *Input Validation*), CppCheck can detect more than 80% of them. On the other hand, the number of FPs is higher than the number of TPs. As a consequence, the precision is between 0.46 and 0.47. Flawfinder does not have good results for these categories. The recall is 0.3837 for *Input Validation* and 0.4278 for *Memory Management*, and the precision is 0.0192 for the latter and 0.0113 for the prior. Except for the recall for CppCheck, all the values for these two categories are low (below 0.5). As a consequence, we can say that neither CppCheck nor Flawfinder can perform acceptably in the detection of these categories of vulnerabilities.

Table 4.3: Results of the SATs - Vulnerability Categories - CppCheck

	CppCheck			
	TN	FP	FN	TP
Memory Management	96.4766% (9,748)	3.5234% (356)	13.6111% (49)	86.3889% (311)
Input Validation	97.8080% (9,995)	2.1920% (224)	17.5510% (43)	82.4490% (202)
Permission	98.6678% (10,147)	1.3322% (137)	32.2222% (58)	67.7778% (122)
Cryptography	99.9139% (10,446)	0.0861% (9)	22.2222% (2)	77.7778% (7)
Data Protection	99.1162% (10,318)	0.8838% (92)	7.4074% (4)	92.5926% (50)
System Configuration	99.8468% (10,427)	0.1532% (16)	28.5714% (6)	71.4286% (15)
File Management	99.6457% (10,405)	0.3543% (37)	9.0909% (2)	90.9091% (20)
Coding Practices	99.9809% (10,454)	0.0191% (2)	37.5000% (3)	62.5000% (5)

Table 4.4: Results of the SATs - Vulnerability Categories - Flawfinder

	Flawfinder			
	TN	FP	FN	TP
Memory Management	21.9913% (2,222)	78.0087% (7,882)	57.2222% (206)	42.7778% (154)
Input Validation	19.6203% (2,005)	80.3797% (8,214)	61.6327% (151)	38.3673% (94)
Permission	30.4454% (3,131)	69.5546% (7,153)	56.6667% (102)	43.3333% (78)
Cryptography	42.6791% (4,462)	57.3219% (5,993)	22.2222% (2)	77.7778% (7)
Data Protection	30.8069% (3,207)	69.1931% (7,203)	50.0000% (27)	50.0000% (27)
System Configuration	65.2782% (6,817)	34.7218% (3,626)	80.9524% (17)	19.0476% (4)
File Management	98.8795% (10,325)	1.1205% (117)	63.6364% (14)	36.3636% (8)
Coding Practices	58.8753% (6,156)	41.1247% (4,300)	75.0000% (6)	25.0000% (2)

Table 4.5: Performance of the SATs

		Precision	Recall	F-Measure
CppCheck	Memory Management	0.4663	0.8639	0.6056
	Input Validation	0.4742	0.8245	0.6021
	Permission	0.4710	0.6778	0.5558
	Cryptography	0.4375	0.7778	0.5600
	Data Protection	0.3521	0.9259	0.5102
	System Configuration	0.4839	0.7143	0.5769
	File Management	0.3509	0.9091	0.5063
	Coding Practices	0.7143	0.6250	0.6667
Flawfinder	Memory Management	0.0192	0.4278	0.0367
	Input Validation	0.0113	0.3837	0.0220
	Permission	0.0108	0.4333	0.0210
	Cryptography	0.0012	0.7778	0.0023
	Data Protection	0.0037	0.5000	0.0074
	System Configuration	0.0011	0.1905	0.0022
	File Management	0.0640	0.3636	0.1088
	Coding Practices	0.0005	0.2500	0.0009

The categories that CppCheck has the highest recall are *Data Protection* and *File Management*. In both cases, the value is above 0.90. On the other hand, these are also the cases where CppCheck presents the lowest precision values (about 0.35). For each category except *Coding Practices*, CppCheck has a precision that is lower than its overall precision. Also, CppCheck does not present a F-Measure value for any category that is higher than its overall F-Measure. This happens because the number of TPs is smaller (due to a smaller number of instances per vulnerability category), and the number of FPs does not decrease in the same proportion (reported TPs of other vulnerability categories are considered as FPs when a particular category is being analyzed).

Flawfinder has a good recall of 0.7778 for *Cryptography*, although it ties with CppCheck. On the other hand, it has one of the worst precision (0.0012) and F-Measure (0.0023) results for this category, as a consequence of the high number of FPs. Due to the highest value of precision (0.0640), Flawfinder presents the highest value of F-Measure (0.1088) for the *File Management* category. These results are better than the overall Flawfinder results. Considering recall, Flawfinder performed better in four categories (*Memory Management*, *Permission*, *Cryptography*, and *Data Protection*) than when not considering categories.

Answering the **RQ1** on whether *SATs are effective in detecting different types of software vulnerabilities in large-scale software projects?*, we conclude that none of the two SATs can be trusted to effectively detect vulnerabilities of any category. This is evident considering the F-Measure results, which ranged from 0.50 to 0.67 for CppCheck and from 0.01 to 0.11 for Flawfinder.

Key Observations:

- Some vulnerabilities are fixed without code changes
- None of the SATs presents good results, but CppCheck is overall better than Flawfinder
- The SATs present better results for some categories, but such categories have few reported vulnerabilities

4.3 Threats to Validity

Internal Validity refers to the possibility of having unanticipated relationships. Although we have a dataset of actual vulnerabilities, we cannot assure that the source code does not have other vulnerabilities. Thus, some of the alerts classified as FPs can be actual vulnerabilities. As the number of alerts is large, we cannot validate them manually. The use of other SATs could help deciding if each alert is a vulnerability or not, using a simple majority approach.

External Validity refers to the ability to generalize the results. In this case, it refers to the used SATs. CppCheck and Flawfinder use mainly syntactic pattern matching and data flow analysis techniques to detect vulnerabilities. SATs with different techniques should be used to enrich the results. SATs with different techniques other than the ones used here may perform better in some of the vulnerability categories. Although other open-source SATs were analyzed, none of them could be run in the Mozilla repository.

4.4 Summary

This chapter studied the use of open-source SATs in large C/C++ projects. The evaluation was performed considering 2,441 Mozilla vulnerabilities from 2000 to 2016. Such vulnerabilities are fixed through 1,590 patches/commits, being 1,017 of them the target of our analysis (as they change C/C++ files). The SATs are evaluated in the complete set of vulnerabilities and per vulnerability category.

Results show that none of the used SATs present good results in any case, although CppCheck performs better than Flawfinder. When vulnerability categories are considered, the overall results are similar (although the tools are able to identify almost all vulnerabilities for the categories with fewer samples). The best F-Measure is 0.71, obtained by CppCheck when all the vulnerabilities are considered, which is not an acceptable value for most development teams. Nevertheless, this SAT may help teams identifying vulnerabilities at the cost of analyzing a high number of FPs.

Applying SATs in a mature and large project may lead to a large number of false alerts, in addition to not detecting many vulnerabilities. For a better understand-

ing of vulnerabilities and their fixes, in the next chapter, we present a detailed analysis of a key type of vulnerability in systems software: *buffer overflow*. Both the SAT alerts and the SMs support the analysis, and vulnerability fixes are analyzed and classified using the Orthogonal Defect Classification (ODC). The goal is to understand how vulnerability fixes impact the code, shedding some light on the causes why SATs do not work in some cases.

Chapter 5

Understanding Buffer Overflow Vulnerabilities

Vulnerabilities in projects developed in C and C++ account for 52% of the known vulnerabilities in open source software [Source, 2021]. Among these vulnerabilities, **improper use of memory**, which may lead to **buffer overflows**, is the most frequent type in these programming languages [Source, 2021].

Techniques to detect vulnerabilities are usually weaker for buffer overflow vulnerabilities [Kratkiewicz and Lippmann, 2005]. To improve the current situation, we need to better understand how buffer overflow vulnerabilities are fixed and what are the capabilities and limitations of using SATs and SMs to detect such weaknesses. SATs indicates potential problems in the source code (as discussed in Chapter 4), while SMs reveal code structural characteristics. This helps researching new approaches for vulnerability detection or improve the existing ones (*e.g.*, by creating new detection rules for SATs). For instance, to support our work on vulnerability detection using ML (Chapter 6) and on the characterization of code units from a security perspective (Chapter 7).

This chapter studies the **characteristics of code with buffer overflow vulnerabilities and of the fixes applied to remove them**. To support the study, we rely on SMs and SAT alerts as they are frequently used to collect information about the quality of the code during the development phase [Jiang et al., 2008] and can be easily extracted from the code under development. Furthermore, vulnerability fixes are analyzed to understand how software developers remove the buffer overflow vulnerabilities from the source code. Although this type of study is also important for other types of vulnerabilities, they should be treated independently, as different conclusions may be reached depending on the specific characteristics of each type.

The analysis follows three complementary directions: *i)* **study the changes in the source code when fixing buffer overflow vulnerabilities** by applying the Orthogonal Defect Classification (ODC); *ii)* **study the SAT alerts in the vulnerable and neutral versions of code units** by running two widely known C/C++ SATs (CppCheck and Flawfinder); and *iii)* **understand the eventual correlation of SMs with the existence of buffer overflow vulnerabilities** by comparing their varia-

tion between the vulnerable and neutral versions of the code. In practice, the goal is to cast light on the following RQs:

- **RQ1:** What are the main changes in the code when fixing buffer overflow vulnerabilities?
- **RQ2:** What are the differences between SAT alerts reported before and after fixes?
- **RQ3:** What is the impact of buffer overflow vulnerability fixes on SMs that portray code characteristics?

The analysis follows three complementary directions: *i)* **study the changes in the source code when fixing buffer overflow vulnerabilities** by applying the Orthogonal Defect Classification (ODC); *ii)* **study the SAT alerts in the vulnerable and neutral versions of code units** by running two widely known C/C++ SATs (CppCheck and Flawfinder); and *iii)* **understand the eventual correlation of SMs with the existence of buffer overflow vulnerabilities** by comparing their variation between the vulnerable and neutral versions of the code. Results show that there is no correlation between SMs and the existence of buffer overflow in the code. Also, most of these vulnerabilities are fixed by adding a checking statement before using a memory space.

The rest of this chapter is organized as follows. Section 5.1 describes the approach and the experiments conducted. The results are presented and discussed in Section 5.2, including the ODC classification and the analysis of the SAT alerts and SMs of the vulnerability fixes. Section 5.3 discusses the limitations of the work and highlights the threats to validity of the results. Finally, Section 5.4 summarizes the chapter.

5.1 Vulnerabilities and Approach

The vulnerabilities (*dataset slice*) in the study are from three C/C++ open-source projects: Linux Kernel, Mozilla, and Xen. From the five projects in the dataset (see Chapter 3), we selected the three with the largest codebase and with the highest number of known vulnerabilities. We analyze the vulnerable version and the neutral version (after a code fix) of a set of 159 code units of the selected projects (also corresponding to 159 vulnerabilities).

The approach followed, depicted in Figure 5.1, is composed of six steps: *i)* select several representative software projects, from a security perspective; *ii)* retrieve vulnerability metadata; *iii)* collect source code versions; *iv)* classify the vulnerabilities using ODC; *v)* collect and analyze SATs alerts; and *vi)* collect and analyze SMs. Note that our approach is generic and can be applied to other vulnerabilities and other contexts (*e.g.*, vulnerabilities in web applications or specific for a project). An example of its use is an analysis of vulnerabilities of the deep learning library TensorFlow presented in [Filus and Domańska, 2023]. Each step is detailed in the following sections.

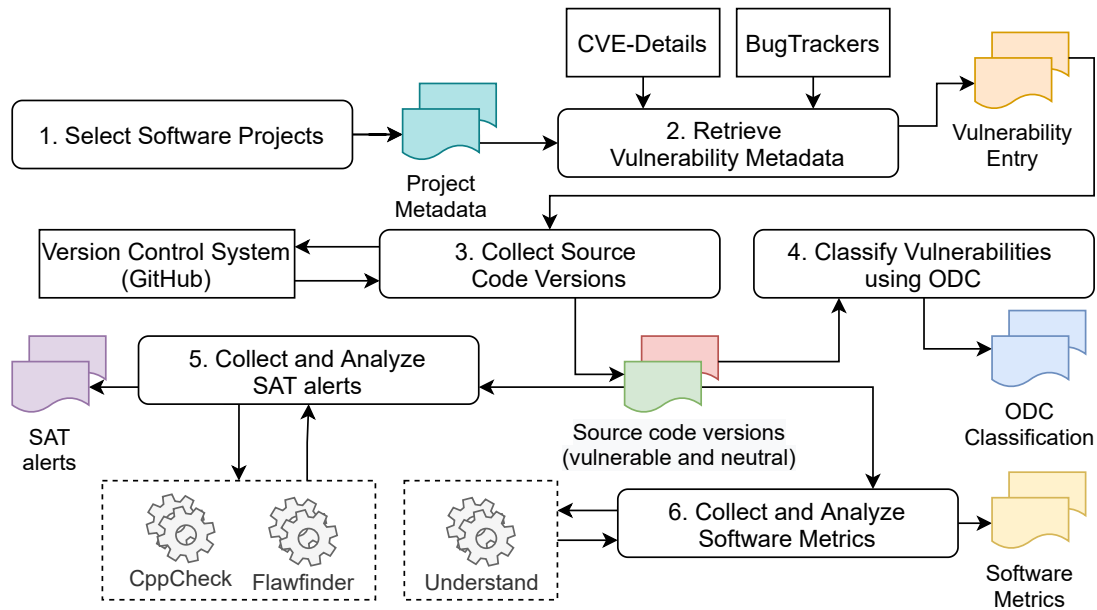


Figure 5.1: Approach for analysis of buffer overflow vulnerabilities

5.1.1 Select Software Projects

The projects selected (Linux Kernel, Mozilla, and Xen) are representative from a security perspective as they are widely used and were frequently the target of security attacks during a long period of time, as it can be seen in CVE Details, the three projects are listed as the top-50 vendors with distinct vulnerabilities¹. The other two projects in the dataset (glibc and httpd) are not considered in this study, as they are small projects with a small number of reported vulnerabilities, thus not allowing to draw relevant observations and conclusions.

CVE Details provides several pieces of information about known vulnerabilities, including the CVSS, the impact, the vulnerability type (which can have multiple values), and the Common Weakness Enumeration (CWE) (although not all vulnerabilities have a CWE assigned). As we are dealing particularly with “Buffer Overflow”, we consider only the records with the identifier *CWE-119: Buffer Overflow* [MITRE, 2006b], which represent between 12.2% to 14.2% of all vulnerabilities in each of the three projects. From the vulnerabilities with a *CWE-119* identifier, we only selected the ones whose type attribute includes the value “Overflow”, which represent between 2.3% to 11.3% of the vulnerabilities in each project. This is shown in Table 5.1, which provides a quantitative summary of the vulnerabilities of the three projects.

Although the number of records for analysis is not large (only 159), it is acceptable as we are dealing with a single type of vulnerability. The vulnerabilities were detected over a considerable period of time (from 2000 to 2016) in several versions of the three C/C++ projects, which have different purposes and functionalities (an operating system, a browser, and a hypervisor). This results in diverse samples with potentially different root causes, allowing to draw relevant observations and meaningful conclusions.

¹<https://www.cvedetails.com/top-50-vendors.php>

Table 5.1: Number of Vulnerabilities: **A)** all vulnerabilities, **B)** vulnerabilities with the CWE-119 identifier, **C)** CWE-119 vulnerabilities with “Overflow” in vulnerability type

	Linux Kernel	Mozilla	Xen	Total
A	867 (100.0%)	2441 (100.00%)	148 (100.0%)	3456 (100.0%)
B	123 (14.2%)	336 (13.8%)	18 (12.2%)	477 (13.8%)
C	98 (11.3%)	56 (2.3%)	5 (3.4%)	159 (4.6%)

5.1.2 Retrieve Vulnerability Metadata

The information about the vulnerabilities was collected from CVE Details. Each vulnerability entry has a unique identifier called CVE-ID, which is composed of the prefix “CVE”, the year in which the vulnerability was reported, and a sequential number. Each CVE-ID includes basic information about the vulnerability, such as the CVSS, the impact, the vulnerability type (which can have multiple values), the CWE, and a table with all software versions affected by that vulnerability. For some vulnerabilities, information on how they can be exploited is also included.

Each CVE-ID was complemented with data from BugTrackers, which are software applications that are used to manage the issues reported for a project (*e.g.*, Mozilla uses Bugzilla as BugTracker). BugTrackers include key information about the specific commit where a vulnerability was fixed.

5.1.3 Collect Source Code Versions

The source code of the projects considered in this study is either stored in GitHub² or in a mirror of the repository available in GitHub. Knowing the commit that fixes a vulnerability (from the BugTracker entries), we obtained the neutral version of the source code (with the vulnerability fixed) and its previous version (the vulnerable one). With these, we can manually analyze the code and run the tools to extract the required information (*e.g.*, SATs alerts and SMs).³

5.1.4 Classify Vulnerabilities using ODC

The next step is to classify the vulnerabilities using ODC [Chillarege et al., 1992], which is a systematic approach to classify defects identified in a software system. It is widely used for root-cause analysis and defines several attributes to be filled when the defect is open and closed based on a predefined set of values. ODC is

² <https://github.com>

³ Although the static data (SAT alerts and SMs) and the fixing commit hash are already in the dataset (as presented in Chapter 3), the source code for the changes is not. Hence, we retrieved the source code from the repositories to apply ODC and used the static data from the dataset.

considered orthogonal as attributes cannot have more than one value assigned. The attributes registered when a defect is open include: *i*) activity, *ii*) trigger, and *iii*) impact. When the defect is closed, the following attributes are captured: *i*) target, *ii*) defect type, *iii*) qualifier, *iv*) source, and *v*) age.

Although ODC includes several attributes, we focus on two of them: defect type and qualifier. We do not consider the other ones as they bring no relevant information towards identifying potential improvements to vulnerability detection (*e.g.*, *activity* characterizes the moment that the vulnerability was discovered, but we are focused on how it was fixed and not when). We use the definition of defect type from [Chillarege et al., 1992], and the possible values are:

1. **Assignment/Initialization:** a problem related to an assignment of a variable or no assignment at all
2. **Checking:** a problem with conditional logic (*e.g.*, condition in a if-clause or in a loop)
3. **Timing:** a problem with serialization of shared resources
4. **Algorithm/Method:** a problem with implementation that does not require a design change to be fixed
5. **Function:** a problem that needs a reasonable amount of code to be fixed due to incorrect implementation or no implementation at all
6. **Interface:** a problem in the interaction between components (*e.g.*, parameter list)

As for the qualifier, we can have:

1. **Missing:** new code needs to be added to fix the defect
2. **Incorrect:** the code is incorrectly implemented and needs adjustment to fix the defect
3. **Extraneous:** unnecessary code needs to be removed to fix the defect

As GitHub highlights the code changes from the vulnerable to the neutral version, we have the information needed to classify these two attributes for each vulnerability. However, because the classification has to be done manually, we decided to have it done by two different researchers to get more accurate results. To have a common baseline for the two researchers, they started by analyzing and classifying together a sub-set of 30 vulnerabilities (out of the 159), which allowed discussing divergences in the approach and reaching a common rationale. This is very important as the experience of the two researchers is different. While one is a post-doctoral researcher who has worked with ODC before, the other is a Ph.D. student⁴ having the first practical experience with ODC in this work. After

⁴ The author of this thesis was the Ph.D. student working with ODC.

that step, each researcher classified the remaining 129 vulnerabilities individually, and the results were merged and consolidated at the end (the divergences in the classification were discussed to come up with a final classification).

As the fix of some vulnerabilities involves several code changes (in more than one block of code in a file or even in several files), each vulnerability may lead to a different number of ODC classifications by different researchers. Hence, the total number of ODC classifications (as presented in subsection 5.2.1) is larger than the total of vulnerabilities analyzed (Table 5.1).

To assess if the classification of the researchers is consistent, we calculated the Inter-Rater Reliability (IRR) metric Cohen's Kappa [Cohen, 1960] on the items classified separately by the two researchers. To interpret the metric, we use the *Landis and Koch* interpretation [Landis and Koch, 1977]: *a*) less than 0: no agreement; *b*) 0–0.20: slight agreement; *c*) 0.21–0.40: fair agreement; *d*) 0.41–0.60: moderate agreement; *e*) 0.61–0.80: substantial agreement; and *f*) 0.81–1.0 almost perfect.

5.1.5 Collect and Analyze SAT Alerts

As the dataset already contains alerts from two SATs (CppCheck and Flawfinder), these data were used. If the alerts were not available, we would need to select relevant SATs and run them in the source code to obtain them. By analyzing the differences in terms of the alerts reported in the two versions of each code unit, we can study the ones that disappear due to a vulnerability fix (when compared with the vulnerable version) and the ones that appear in a vulnerability fix (which can lead to other vulnerabilities in the future).

5.1.6 Collect and Analyze Software Metrics

From that dataset, we used the file-level metrics for the Linux Kernel, Mozilla, and Xen projects (for the 159 buffer overflow vulnerabilities)⁵. We considered only the SMs at the file level, as most of the C/C++ code in these projects is not structured in classes. Also, for our analysis, the function metrics would not provide more information compared to file metrics, as some files are not structured in functions (such as scripts or header files). The SMs are detailed in Appendix B.

5.2 Results and Discussion

This section presents and discusses the results. First, we present the ODC classification of the buffer overflow vulnerabilities. Then, we analyze the main differences in terms of the SAT alerts raised in vulnerable and patched versions. Finally, we discuss the impact of vulnerability fixes in the SMs.

⁵ Tools such as the SciTools Understand [SciTools, 2011] can be used to calculate the SMs in scenarios where a dataset like our is not available.

5.2.1 Main Code Changes when Fixing Vulnerabilities

The 159 buffer overflow vulnerabilities were classified according to the two ODC attributes (defect type and qualifier) by the two researchers. The subset of 30 vulnerabilities classified by the two researchers together (in a meeting that lasted a bit more than one hour) led to a total of 35 classifications, as some vulnerabilities were classified with more than a pair of type/qualifier values. Although ODC is an orthogonal classification (meaning that an attribute should not have more than one value assigned), this happens in our case as a vulnerability fix may require one or more independent code changes. In other words, several block changes may be needed in a single fix, which leads to more than one ODC classification per vulnerability (in practice, we can say that several code weaknesses/faults lead to one vulnerability).

The 129 vulnerabilities that were classified separately resulted in 167 and 177 classifications by each researcher (each one spent between 5 to 6 hours in this classification step). With these data, we computed the IRR Cohen's Kappa metric to assess the consistency between the classifications. We obtained the following values: *i) defect type*: 0.4476 (moderate agreement), and *ii) qualifier*: 0.3939 (fair agreement). Additionally, we computed the Cohen's Kappa considering both ODC attributes (*defect type + qualifier*) as a single classification. The result is 0.4255, a moderate agreement between the researchers. Although these results do not indicate an excellent agreement, they show a fair to moderate agreement according to *Landis and Koch* interpretation [Landis and Koch, 1977], suggesting that our ODC classification for the buffer overflow vulnerabilities is quite consistent.

The divergences in the classifications were discussed (in a meeting that lasted 2 hours), resulting in 216 classifications (pairs of defect types and qualifiers) for the 159 vulnerabilities. This happens because 35 (22.01%) vulnerabilities were fixed by changing more than one file, as can be seen in the box plot in Figure 5.2, which indicates that some buffer overflow vulnerabilities are due to the interaction of more than one software component. Such interactions clearly make the vulnerability detection process difficult to automate.

A summary of the ODC classification process is shown in Figure 5.3, and the consolidated results are presented in Table 5.2. Regarding the defect type, the most frequent vulnerabilities are from "Checking" (85 cases, 39.35%), followed by "Algorithm/Method" (64 cases, 29.63%), and "Assignment/Initialization" (42 cases, 19.44%). This confirms the observations of Morrison et al. [2018] in their study using ODC+V. Regarding the qualifier, "Incorrect" is the most frequent one (123 cases, 56.94%) followed by "Missing" (88 cases, 40.74%).

Table 5.3 (defect type) and Table 5.4 (qualifier) summarize the results of classification per project. Similar to the overall results, the majority of the classifications belong to the "Checking" defect type, for both Linux Kernel and Mozilla projects. This seems to be an obvious approach to prevent out-of-bound access, but developers still fail to add them to the source code. This probably happens as the developers do not anticipate the need for the "Checking" since it is difficult to consider all possibilities and identify the ones that may lead to a buffer overflow problem. Moreover, the developers are probably not supported by adequate tools

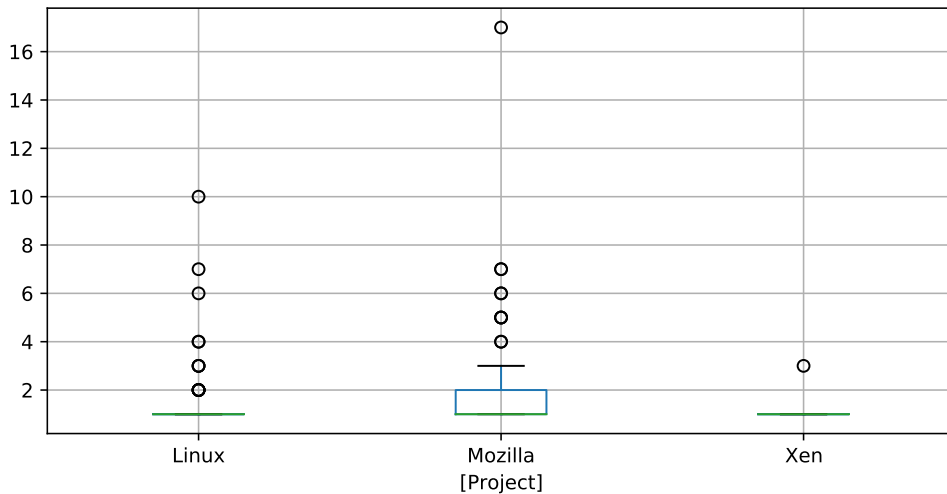


Figure 5.2: Number of changed files per vulnerability fix

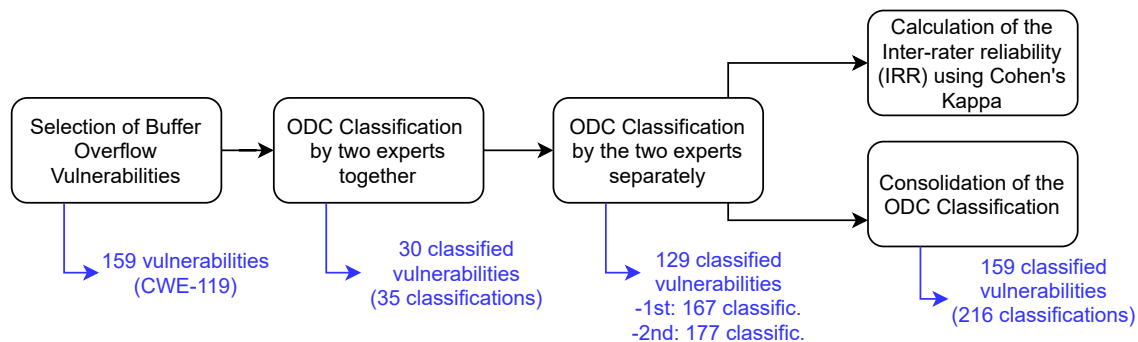


Figure 5.3: Analysis of the buffer overflow vulnerabilities using ODC

Table 5.2: Vulnerability distribution across the ODC Defect Type and the Qualifiers

Defect Type	Incorrect	Missing	Extraneous	Total
Checking	31 (14.35%)	51 (23.61%)	3 (1.39%)	85 (39.35%)
Algorithm/Method	50 (23.15%)	12 (5.56%)	2 (0.93%)	64 (29.63%)
Assignment/Initialization	24 (11.11%)	18 (8.33%)	0 (0.00%)	42 (19.44%)
Interface	16 (7.41%)	3 (1.39%)	0 (0.00%)	19 (8.80%)
Function	2 (0.93%)	4 (1.85%)	0 (0.00%)	6 (2.78%)
Timing	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	123 (56.94%)	88 (40.74%)	5 (2.31%)	216 (100.00%)

that help them identifying or verifying the “Checking” conditions. Another possible reason is the lack of testing skills, such as applying boundary-value analysis when creating the functionality.

Regarding the qualifier attribute, more than half of the Linux Kernel classifications are labeled as “Incorrect” (74 cases, 61.67%), while for the Mozilla project, the number of “Incorrect” (46 cases, 50.55%) and “Missing” (42 cases, 46.15%) are

Table 5.3: Vulnerability distribution across the ODC Defect Type for the projects (Linux Kernel, Mozilla, Xen)

Defect Type	Linux Kernel	Mozilla	Xen
Checking	45 (37.50%)	39 (42.86%)	1 (20.00%)
Algorithm/Method	33 (27.50%)	27 (29.67%)	4 (80.00%)
Assignment/Initialization	27 (22.50%)	15 (16.48%)	0 (0.00%)
Interface	11 (9.17%)	8 (8.79%)	0 (0.00%)
Function	4 (3.33%)	2 (2.20%)	0 (0.00%)
Timing	0 (0.00%)	0 (0.00%)	0 (0.00%)
Total	120 (100.00%)	91 (100.00%)	5 (100.00%)

Table 5.4: Vulnerability distribution across the ODC Qualifier for the projects (Linux Kernel, Mozilla, Xen)

Qualifier	Linux Kernel	Mozilla	Xen
Incorrect	74 (61.67%)	46 (50.55%)	3 (60.00%)
Missing	45 (37.50%)	42 (46.15%)	1 (20.00%)
Extraneous	1 (0.83%)	3 (3.30%)	1 (20.00%)
Total	120 (100.00%)	91 (100.00%)	5 (100.00%)

quite similar. The small number of classifications for the Xen project does not allow further analysis.

The above results help answering the **RQ1**, about *the main changes in the code when fixing buffer overflow vulnerabilities*, and indicate that projects lack mechanisms to verify simple conditional logic (“Checking” defect type) and assignment and initialization of variables (“Assignment/Initialization” defect type). Moreover, some review effort could be beneficial to identify major issues in the implementation (“Algorithm/Method” defect type). In all cases, issues can be either absent (“Missing” qualifier) or poorly implemented (“Incorrect” qualifier) code. At first sight, the detection of some of these cases may be easily automated (*e.g.*, by adding simple rules to SATs), but many others require human intervention or other advanced detection techniques (*e.g.*, in the case of vulnerabilities that occur due to multiple weaknesses in different files). As an example, let’s take a look at the Linux Kernel vulnerability CVE-2014-0049, whose fix can be seen below. It shows the code of the Linux Kernel file `arch/x86/kvm/x86.c` after the fix of CVE-2014-0049.

```
if (vcpu->mmio_cur_fragment >= vcpu->mmio_nr_fragments) {
    // removed due to space constraints
}
```

In this example, the *if-clause* was classified as being “Incorrect” because the operator had to be changed from `==` (equal comparison in C/C++) to `>=` (greater than or equal comparison) to fix the vulnerability. The buffer overflow happened

as the execution of the function was not interrupted when the number of the current fragment exceeded the total number of fragments. This cannot be automated through SAT rules as it requires a complex and semantic interpretation. Although the Linux Kernel development team could create a rule for this specific case, it turns out to be totally context-specific, thus, when used in different contexts, it would lead to a high number of false alarms.

Key Observations:

- Most buffer overflow vulnerabilities are fixed by a simple change in conditional logic (either incorrect or missing), which is not anticipated by developers and not identified by tools
- Some vulnerabilities could have been detected with the use of adequate SATs
- Not all vulnerabilities can have their identification automated as they involve intricate issues in several files
- A manual review process could help identifying vulnerabilities earlier in the SDLC

5.2.2 SAT Alerts Before and After Vulnerability Fixes

The analysis of the SAT alerts started by querying the alerts of two SATs (CppCheck and Flawfinder) of the two versions of each code (vulnerable and neutral) for each buffer overflow vulnerability. With this, we can analyze what has been changed between the vulnerable and the neutral versions. Three cases are possible: *i*) alerts disappearing from the vulnerable version to the neutral version due to the code fix; *ii*) new alerts appearing in the code that fixed the vulnerability; and *iii*) SAT alerts either appearing or disappearing in untouched code (part of the code that was not affected by the fixes). Depending on the techniques used by the SATs (*e.g.*, data flow analysis, taint analysis), some code changes may cause the SATs to raise new alerts in parts of the code that were not touched. Thus, we consider all the alerts, and not only the ones in the changed code. Nevertheless, as most alerts are kept equal from one version to the other, we can filter these out. In practice, we are simplifying the analysis by excluding the alerts that have the same type and are raised in the same (or corresponding) lines of code in both the vulnerable and the neutral versions of each code unit.

Considering the types of SAT alerts for which we identified variations between the vulnerable and the neutral versions of each vulnerability, none appeared in more than one project. For example, the alert type `wcscpy` from Flawfinder is raised in the Mozilla project but not in the Linux Kernel and Xen projects. This happens both for CppCheck and Flawfinder alerts in the different projects. Due to the reduced number of vulnerabilities in Xen, we could not observe much regarding SAT alerts changing due to code fixes: only one alert with CppCheck (`doubleFree`) and none with Flawfinder.

Table 5.5: Number of alerts (minimum and maximum among all commits) reported by the SATs for the complete code-base in all vulnerabilities

	CppCheck	Flawfinder
Linux Kernel	35,110 (min.)	32,157 (min.)
	79,135 (max.)	55,930 (max.)
Mozilla	24,660 (min.)	5,755 (min.)
	45,769 (max.)	22,543 (max.)
Xen	4,545 (min.)	3,079 (min.)
	4,717 (max.)	3,216 (max.)

Table 5.6: CppCheck SAT alerts reported in Linux Kernel that changed from the vulnerable to the neutral versions

SAT	Linux Kernel	
	Vulnerable	Neutral
nullPointerRedundantCheck	3	0
unusedStructMember	1	0
Total	4	0

Table 5.5 shows a summary of the minimum and the maximum number of alerts per SAT raised for different commits of each project. For example, Linux Kernel has 98 vulnerabilities analyzed in this study (see Table 5.1), we ran the SATs for those vulnerabilities for different commits and counted all reported alerts per vulnerability for each commit. The smallest number of alerts raised for Linux Kernel by CppCheck belongs to a specific commit and is equal to 35,110 alerts, and the largest number of alerts raised for Linux Kernel by CppCheck belongs to another commit and is equal to 79,135. Overall, CppCheck reports more alerts for the complete codebase of each project than Flawfinder. The project with the largest number of alerts is Linux Kernel (maximum of 79,000 reported by CppCheck). On the other hand, Xen is the project with the smallest number of reported alerts (minimum of 3,000 reported by Flawfinder).

Table 5.6 and Table 5.7 show the CppCheck results for Linux Kernel and Mozilla, respectively. Table 5.8 and Table 5.9 show the Flawfinder results (as this tool defines categories for the types of alerts, they are also included in the tables). As shown, the total number of alerts that vary from vulnerable to neutral versions is very small when compared to the number of alerts raised per project. For example, for the Mozilla project, the total number of Flawfinder alerts varies between 5,755 and 22,543 (Table 5.5) over the 56 commits considered. However, the alerts that changed between the vulnerable and neutral versions of all vulnerabilities are only 21 (Table 5.9). In other words, 21 alerts that were raised in vulnerable versions disappeared when fixes were implemented. On the other hand, 19 new alerts appeared after vulnerability fixes.

Table 5.7: CppCheck SAT alerts reported in Mozilla that changed from the vulnerable to the neutral versions

SAT	Mozilla	
	Vulnerable	Neutral
syntaxError	1	1
toomanyconfigs	0	1
uninitMemberVar	1	0
unusedFunction	1	0
memsetClassFloat	1	0
Total	4	2

Table 5.8: Flawfinder SAT alerts reported in Linux Kernel that changed from the vulnerable to the neutral versions

Category	SAT Type	Linux Kernel	
		Vulnerable	Neutral
Buffer	memcpy	8	5
	char	6	6
	strlen	7	4
	strcpy	6	3
	sprintf	6	1
	strncpy	4	3
	strncat	1	0
Format	syslog	14	14
	printf	0	8
	vsprintf	1	1
Total		53	45

From the 159 vulnerabilities analyzed, only 22 fixes (13.84%) impacted the SAT alerts, but only one impacted the alerts raised by both SATs (CVE-2007-6151). Furthermore, although all projects are written in the same programming languages (C/C++), and the vulnerabilities analyzed are of the same type (CWE-119), very different SAT alerts are raised in the three projects (with no clear overlap), suggesting that the root causes of buffer overflow vulnerabilities differ a lot from each other. Let's analyze a couple of examples.

The following code snippet presents the fix of the Linux Kernel CVE-2010-4527 vulnerability in the file `linux/sound/oss/soundcard.c`. In this case, the vulnerable function `strcpy` has been replaced by `strncpy`, which is also considered vulnerable (line 102). Flawfinder raised alerts in both the vulnerable and the (supposedly) neutral versions, meaning that part of the alerts on the `strcpy` version

Table 5.9: Flawfinder SAT alerts reported in Mozilla that changed from the vulnerable to the neutral versions

Category	SAT Type	Mozilla	
		Vulnerable	Neutral
Buffer	wcscpy	19	0
	wcsncpy	0	19
	wcslen	1	0
Format	fprintf	1	0
Total		21	19

migrated to strncpy version. This fix also changes the strcmp (considered unsafe) to strncmp (line 90), but none of the SATs raised an alert for the unsafe function strcmp.

```
// Line 90
if (strncmp(name, mixer_vols[i].name, 32) == 0) {

    // Unchanged lines: 91-101

    // Line 102
    strncpy(mixer_vols[n].name, name, 32);
```

To replace the vulnerable function strncpy with a safe one, strncpy can be used, as done in CVE-2013-2850 (file drivers/target/iscsi/iscsi_target_parameters.c). Note that the use of vulnerable functions is a known weakness still present in many software systems. In fact, OWASP lists using “Vulnerable and Outdated Components” as one of the top 10 weaknesses that software developers should prevent [Foundation, 2021].

```
strncpy(extra_response->key, key, sizeof(extra_response->key));
```

The following code snippet shows an example of a “Checking” that was missing in the code and that was not detected by any of the SATs. This is part of the vulnerability CVE-2013-1721. In the vulnerable version, only the first part of the condition was included; the second condition has been added to fix the vulnerability by checking the required space needed for a buffer in use.

```
else if (mWritePosition + requiredSpace > mBufferSize ||
mWritePosition + requiredSpace < mWritePosition) // Recycle
```

These results help answering RQ2 about *the difference between SAT alerts reported before and after fixes*. As shown, in most cases, the vulnerability fixes do not change the outcome of the SATs, suggesting a low capability of these tools to detect buffer overflow vulnerabilities.

Key Observations:

- A small number of vulnerabilities are detected by SATs, especially when unsafe functions are used
- Some fixes lead to new SAT alerts, sometimes related to the use of other unsafe functions (*e.g.*, `wcscopy` to `wcsncpy`)
- Not all vulnerabilities can be detected by SATs as they involve the interaction among diverse components
- New SAT rules are needed to detect specific vulnerabilities, in particular, the other related to *checking* conditions

5.2.3 Impact of Vulnerability Fixes on Software Metrics

One of the main arguments in previous works for using SMs to detect software vulnerabilities is that they allow portraying the size and complexity of the source code and, usually, more complex code is more prone to have vulnerabilities [Shin et al., 2011; Walden et al., 2014]. However, this assumption needs to be confirmed in order to gain trust in the use of SMs to detect software vulnerabilities. The 54 SMs used in this study portray different characteristics of the source code, such as volume, coupling, cohesion, and complexity.

Table 5.10 presents the top 10 of the SMs that changed more frequently when buffer overflow vulnerabilities were fixed, which are mostly volume metrics. For example, considering the Linux Kernel project, we can observe that the *CountLine* metric changed for 68 files out of a total of 75 files modified to fix 98 vulnerabilities. The *CountLine* metric indicates the number of physical lines in a file. As shown, nine out of the ten metrics are the same for the three projects. They either reflect the size of the code (*CountLine*, *AltCountLineCode*, *CountLineCode*, *CountStmt*, *CountLineCodeExe*, *CountSemicolon*, and *CountStmtExe*) or its complexity (*SumCyclomaticStrict* and *SumCyclomaticModified*). Table B.1 in the Appendix B details all the SMs presented in this section.

Although this suggests a correlation between vulnerability fixes and the value of some metrics, there may not be causality. In fact, a detailed analysis of the values of volume (*e.g.*, lines of code) and complexity (*e.g.*, McCabe cyclomatic complexity) metrics allows observing that the value of most metrics increases when a buffer overflow vulnerability is fixed. This is confirmed by the ODC analysis, in which 40.74% of the classifications showed that some code had to be added to fix a vulnerability. The problem is that this may go against the assumption that smaller and simpler code is less prone to be vulnerable.

Moreover, although the value of some metrics is frequently varying from the vulnerable to the neutral versions, it does not say much about the potential existence of vulnerabilities. For example, the metric *Henry Kafura Size (HK)*, which was proposed by Henry and Kafura [1981], also appears as being frequently impacted

Table 5.10: Top 10 SMs impacted by the vulnerability fixes per project (more than 10 items listed as Xen ties in some SMs)

SM	Linux Kernel	Mozilla	Xen
CountLine	68/75 (90.7%)	44/51 (86.3%)	6/6 (100.0%)
AltCountLineCode	66/75 (88.0%)	43/51 (84.3%)	6/6 (100.0%)
CountLineCode	59/75 (78.7%)	38/51 (74.5%)	5/6 (83.3%)
CountStmt	56/75 (74.7%)	34/51 (66.7%)	5/6 (83.3%)
CountStmtExe	53/75 (70.7%)	27/51 (52.9%)	5/6 (83.3%)
CountSemicolon	53/75 (70.7%)	32/51 (62.7%)	5/6 (83.3%)
CountLineCodeExe	53/75 (70.7%)	33/51 (64.7%)	5/6 (83.3%)
HK	49/75 (65.3%)	-	4/6 (66.7%)
SumCyclomaticStrict	46/75 (61.3%)	24/51 (47.1%)	4/6 (66.7%)
SumCyclomaticModified	42/75 (56.0%)	23/51 (45.1%)	4/6 (66.7%)
SumCyclomatic	-	23/51 (45.1%)	4/6 (66.7%)
AltCountLineBlank	-	-	4/6 (66.7%)
CountLineBlank	-	-	4/6 (66.7%)

by vulnerability fixes (in fact, it is among the metrics whose values vary more). HK is the result of the multiplication of three other metrics, being two of them squared ($length * (FanIn * FanOut)^2$, where *length* is a volume metric, such as LOC). Consequently, small changes in the code can result in a large value variation for *HK*, but we cannot assure that is not strictly related to vulnerability fixes (in fact, we can observe similar variations across different versions of the same file, even when no vulnerabilities are being dealt with).

Table 5.11 shows the SMs that remain unchanged due to vulnerability fixes per project. For example, the SM *MaxNesting* has the same value before and after fixing the vulnerability in all the analyzed files for Mozilla and Xen projects. Among these, there is one, *CountDeclClass* (number of classes), that never varies at all (in italic in the table). Clearly, these are metrics that cannot help in the detection of software vulnerabilities.

These results help answering **RQ3** about *SMs changing when buffer overflow vulnerabilities are fixed*. Although most of the fixes lead to changes in SMs, metrics are not more impacted by fixes than by other code improvements. In fact, there is no clear causality between the value of a metric and the existence of a vulnerability. Hence, no SM can be used to detect the presence of buffer overflow vulnerabilities. This has been confirmed by other works that use SMs to detect software vulnerabilities [Medeiros et al., 2020; Shin and Williams, 2013].

Table 5.11: Unchanged SMs in the vulnerability fixes per project

Project	Unchanged SMs
Linux Kernel	<i>CountDeclClass</i> , DIT, NOC, CBC, RFC, CBO, LCOM
Mozilla	MaxNesting, CountStmtEmpty, <i>CountDeclClass</i> , AvgLineBlank, AltAvgLineBlank
Xen	NOC, DIT, CBC, RFC, CBO, AltAvgLineBlank, RatioCommentToCode, MaxNesting, CountStmtEmpty, CountLinePreprocessor, CountDeclFunction, <i>CountDeclClass</i> , AvgLineComment, AvgLineCode, AvgLineBlank, AvgLine, AvgEssential, AvgCyclomaticStrict, AvgCyclomaticModified, AvgCyclomatic, AltAvgLineComment, LCOM

Key Observations:

- Most vulnerability fixes add code to the codebase, leading to an increase in the value of the metrics that are related to volume and complexity
- Causality between vulnerability fixes and the variation in metrics cannot be established
- Code changes related to vulnerability fixes cannot be easily distinguished from other improvements using software metrics
- SMs are not good indicators of the existence of vulnerabilities, in particular buffer overflow, but probably can be used to indicate less trustworthy code units

5.3 Threats to Validity

This section discusses the threats to the validity of the approach and the results obtained. The threats are mainly related to the projects considered, the ODC classification, the SATs used, and the number and types of vulnerabilities analyzed.

External Validity refers to the ability to generalize the results beyond the experiment settings. The study considers a single type of vulnerability: buffer overflow. Although it is the most relevant vulnerability type for C/C++ projects, other vulnerabilities (*e.g.*, related to improper or lack of input validation) are not considered and may lead to different observations and conclusions as the SAT alerts and SMs may vary.

All the projects analyzed in this study are developed in C/C++ and have a large code-base. Buffer overflow vulnerabilities are more relevant for C/C++ projects but not limited to them. Hence, some key observations may not be the same for projects in other programming languages and with different sizes. Nevertheless, the observations are still relevant as buffer overflow is still one of the most frequent issues in many programming languages and has a severe impact on software security, particularly in C/C++ projects that compose many essential and highly used software projects.

The different levels of experience of the researchers and the complexity of some of the fixes may lead to different classification results. We tried to mitigate this by performing an initial joint classification effort before the individual ones. Moreover, individual classifications were discussed in a consensus meeting to reach a final agreed classification. However, both researchers may have misclassified, in the same way, some vulnerabilities. To mitigate this issue in the future, more experts can be asked to perform the classification.

Internal Validity refers to the possibility of having unanticipated relationships. The number of SATs used for the analysis of alerts is limited to two. Nevertheless, these two SATs (CppCheck and Flawfinder) are widely used and have a high number of rules to detect software issues and vulnerabilities. Hence, they provide relevant input for the analysis.

Although the main objective of the selected tools for this study is to detect vulnerabilities, we used them to characterize only buffer overflow vulnerabilities. Thus, another limitation of the study is related to the limited techniques used for the characterization of the buffer overflow vulnerabilities, as other software vulnerability detection techniques, in particular, dynamic techniques such as SPT and fuzzing, could reveal additional characteristics of the buffer overflow vulnerabilities. Nevertheless, due to the project characteristics and the time span of the vulnerabilities, it would not be doable to apply techniques like that in the dataset considered for this study.

Construct Validity refers to the 159 vulnerabilities analyzed to validate our study. This is not a large number, but the diversity of the causes and the representativeness of the projects support some relevant observations. Moreover, most of the analysis involves manual validation of the vulnerability fixes. Hence, performing this study in a larger dataset would be even more time-consuming. Automating the ODC classification would be an option (such as it was performed in other studies such as [Lopes et al., 2020]). However, a manual review would still be needed to validate the classification and fix the incorrect ones.

5.4 Summary

In this chapter, we analyzed a set of buffer overflow vulnerabilities to study potential ways to improve vulnerability detection, either by improving existing techniques or devising new ones. The vulnerabilities of three open-source C/C++ projects (Linux Kernel, Mozilla, and Xen) were used in the analysis. Each vulner-

ability was classified using ODC. Moreover, the SAT alerts and SMs were analyzed and compared for both the vulnerable and neutral versions.

Results show that most of the vulnerable code units are labeled with ODCs defect types *checking* and *algorithm/method*. On the other hand, SATs lack rules to detect most vulnerabilities, in particular missing or incorrect checking logic. Also, we could not find any causality between buffer overflow vulnerability fixes and the value of SMs.

Considering the low performance of SATs observed in Chapter 4 and the observations in this chapter, next present two studies in an attempt to assess the potential use of ML with static code features to detect vulnerabilities. In the first, we use classical ML algorithms are used to predict the presence of vulnerabilities in files. In the second, we follow an approach based on deep learning to detect vulnerable functions grounded on DGCNN and VGG networks.

Chapter 6

Detecting Software Vulnerabilities with Machine Learning

Machine Learning (ML) is being widely used in software engineering, including for cybersecurity [Kurtz, 2019]. Some studies have used SMs as features for ML algorithms to detect defects in the source code [Catal and Diri, 2009; Prasad et al., 2015], and a similar rationale has also been considered to detect vulnerabilities [Alves et al., 2016b; Medeiros et al., 2017; Walden et al., 2014]. The problem is that there is insufficient evidence of the performance and practical applicability of ML (and different types of features) in this context.

This chapter presents two studies that use ML algorithms to detect vulnerable code units. In the first one, we evaluate the possibility of using SMs and SAT alerts to detect software vulnerabilities by means of classical ML algorithms (Decision Tree (DT), Random Forest (RF), Extreme Gradient Boosting (XGB), and Bagging). The Mozilla project is used in this evaluation. The focus is not only on the ability to predict if a file is vulnerable (binary prediction) but also on the prediction of the vulnerability category (multiclass prediction).

Considering more modern algorithms, the second study uses Deep Graph Convolution Neural Network (DGCNN) to detect vulnerable functions. The process is based on MAGIC [Yan et al., 2019], an approach used to detect and classify malware programs in one malware category (multiclass classification). The Control Flow Graph (CFG) of C functions of the Linux Kernel project is explored to extract customized memory management-related features. In addition to the traditional SMs, such features are used as input for deep learning. Note that we use a project different from the first study (Mozilla), as more vulnerabilities were available for the Linux Kernel at the time of the work.

The remaining of this chapter is organized as follows. Section 6.1 presents the study with classical ML algorithms with SAT alerts and SMs as features. Section 6.2 presents the study with DGCNN. The threats to validity are discussed in Section 6.3. Finally, Section 6.4 summarizes the main findings of the two studies.

6.1 Classical Machine Learning

This study aims at validating the hypothesis of **combining alerts of multiple SATs with SMs as features for ML algorithms to predict software vulnerabilities** in large software projects. In particular, we aim at answering the following RQs:

- **RQ1:** Can alerts from several SATs be combined to predict vulnerable code using ML algorithms?
- **RQ2:** Can SAT alerts be complemented with SMs to improve vulnerability detection using ML algorithms?
- **RQ3:** Do ML algorithms perform better when considering vulnerabilities split per category?

In practice, our contribution is two-fold. We present an experiment on the use of **SAT alerts and SMs as inputs (features) for ML algorithms to predict software vulnerabilities**. Four well-known ML algorithms are used in the experiments: DT, RF, XGB, and Bagging. Based on the observations, we analyze the SMs of the vulnerability fixes aiming at identifying **software characteristics that help or make it difficult to create ML models with good performance**. Results show that ML algorithms create better models using SMs than using SAT alerts, although neither of them achieves good precision and recall at the same time.

The remainder of this section is organized as follows: Section 6.1.1 presents the approach followed in this work. The results of the binary classification, binary classification per category, and multiclass classification are in Section 6.1.2. An analysis of source code characteristics with regards to the SMs is presented in Section 6.1.3.

6.1.1 Vulnerabilities and Approach

The *dataset slice* for this study is based on the vulnerabilities of the Mozilla project reported from 2000 to 2016¹. Table 6.1 shows the number of vulnerabilities per category and the number of files that were changed to fix those vulnerabilities. Note that fixing a vulnerability may require modifying several files, and a file may have several vulnerabilities.

Using the data extraction features mentioned in Chapter 3, we extracted a tabular dataset for training and testing the ML models. Figure 6.1 provides a simplified view of its organization. Each sample corresponds to a file in a specific version, and a version corresponds to a GitHub commit. Hence, a file can be identified as vulnerable in one commit and as non-vulnerable (neutral) in another one (SATs alerts and SMs were collected for both the vulnerable and the neutral versions of each file). This is exactly what happens in the hypothetical `file1` of Figure 6.1. This file is non-vulnerable in commit `v1` and vulnerable in commit `v2`.

¹ At the time of this study, the dataset contained only vulnerabilities until 2016.

Table 6.1: Vulnerabilities and number of fixed files per vulnerability categories in the Mozilla dataset. Permission category is the combination of three OWASP groups: Authentication and Password Management, Session Management, and Access Control

Category	# Vuln. (%)	# Files (%)
Memory Management	596 (24.42%)	603 (20.36%)
Input Validation	318 (13.03%)	399 (13.48%)
Permission²	177 (7.25%)	251 (8.48%)
Data Protection	65 (2.66%)	64 (2.16%)
Coding Practices	31 (1.27%)	12 (0.41%)
Cryptography	18 (0.74%)	9 (0.30%)
System Configuration	16 (0.66%)	36 (1.22%)
File Management	9 (0.37%)	23 (0.78%)
Output Encoding	0 (0%)	0 (0%)
Error Handling and Logging	0 (0%)	0 (0%)
Communication Security	0 (0%)	0 (0%)
Database Security	0 (0%)	0 (0%)
CWEs not found	1,211 (49.60%)	1,564 (52.82%)
Total:	2,441 (100.00%)	59,151 (100%)

In practice, the features include both SMs and counts of reported alerts per type per file per SAT. The total number of file-level SMs is 54 (as detailed in Appendix B), and there are 228 SAT alerts types for CppCheck and 126 for Flawfinder, corresponding to a total of 408 features. It is essential to notice that, for each file, most of the SAT features have zero as value, as most files do not have alerts of many types. Nevertheless, they are kept as we later use feature selection techniques to remove them. Each vulnerable sample has a label indicating the vulnerability category. As we also perform experiments to predict vulnerabilities regardless of their category, different versions of the tabular dataset were created to be used in the different experiments (as discussed below). The datasets used in this study can be found online in the following URL: <https://eden.dei.uc.pt/~josep/thesis/>

As expected, the *dataset slice* is highly unbalanced, as the number of vulnerable samples (2,441 samples; 0.4%) is much smaller than the number of the non-vulnerable ones (597,259 samples; 99.6%). To create a feasible tabular dataset including both SATs alerts and SMs, we decided to make the 2,441 vulnerable samples to represent 5% of the total number of samples. For this, during the extraction process, random non-vulnerable samples were obtained from the original *dataset slice* to complete the remaining 95% of the samples (in practice, it corresponds to discarding a subset of samples). The repository of source code files is organized into directories, and each file is in a directory. As vulnerable files con-

		Features								Label
		Software Metrics				SATs				
		DIT	NOC	...	CyclCompl	SAT1 alert_type1	SAT1 alert_type2	...	SATn alert_type_n	
Samples	file1_v1	2	1	...	15	1	0	...	1	Non-vulnerable
	file1_v2	2	1	...	24	1	0	...	0	Vulnerable
	file2_v1	3	3	...	55	0	1	...	1	Non-vulnerable
	file_n_vn	3	2	...	33	0	0	...	1	Vulnerable

Figure 6.1: Example of the resulting dataset

centrate more in some directories than others, we used the vulnerable file distribution to select the non-vulnerable samples. For example, if 5% of the vulnerable samples are from the directory *layout*, we select the same proportion of neutral files (5%) from that same directory. The remaining 95% of each group (vulnerable and neutral) are selected in the same manner from the remaining folders. This way, the data remains unbalanced but still representative of the complete data.

To answer the RQs presented before, we designed three experiments:

1. **Binary Classification:** the samples in the dataset are labeled either as vulnerable or as non-vulnerable (neutral), regardless of the vulnerability category (related to RQ1 and RQ2).
2. **Binary Classification per Category:** the samples in the dataset are labeled as vulnerable in one category or as non-vulnerable in that specific category (e.g., *memory management* vulnerability / non-vulnerable in the *memory management* category). In this case, only the three categories with the most significant number of samples are considered (*memory management*, *input validation*, and *permission*), as the small number of samples in the remaining categories is not enough to train and test ML models. In practice, the non-vulnerable instances may be a file without (known) vulnerabilities or with vulnerabilities of a different category (related to RQ3).
3. **Multiclass Classification:** the samples in the dataset are labeled as vulnerable or non-vulnerable, where each vulnerable sample contains an identifier corresponding to the vulnerability category (class). The samples without a known category are labeled with the same identifier. In other words, they are considered as a different category altogether (related to RQ3).

For each experiment, we considered three dataset configurations: *a*) both SAT alerts and SMs are included as features; *b*) only SMs are included as features; and *c*) only SAT alerts are included as features. In practice, each experiment consists of using the three dataset configurations to train and test different ML models to predict if samples are vulnerable or not. For that, we considered the four algorithms (DT, RF, XGB, and Bagging) that achieved the best results in a set of

Table 6.2: Algorithms and Techniques

Parameter	Values
Algorithms	DT, RF, XGB, Bagging
Feature Selection	Variance, Correlation, Variance+Correlation, Principal Component Analysis (PCA)
Sampling	No sampling technique, Random undersampling, Random oversampling, Random Undersampling + Random Oversampling

preliminary tests, and diverse feature selection and sampling techniques widely used in the ML domain. While the feature selection techniques filter the features to reduce the complexity of the dataset for the ML algorithms, the sampling techniques aim to balance the classes to have the same proportions (vulnerable and non-vulnerable samples). A summary of the algorithms and techniques can be seen in Table 6.2.

All the experiments are performed using Propheticus [R. Campos et al., 2019], a tool to automate the execution of ML algorithms based on the Python library scikit-learn [Pedregosa et al., 2011]. As hyperparameter values for each algorithm, we used the default ones provided by scikit-learn. For the RF, XGB, and Bagging algorithms, the number of estimators is defined as 100.

To evaluate the performance of each ML instance, we need to split the dataset into a training subset and a testing subset. While the former is used by the ML algorithm to train the model, the latter is used to evaluate its performance (*i.e.*, if the model can predict the actual class of a sample). The technique used for this evaluation is Cross Validation (CV), in which the original set is divided into k subsets, and $k - 1$ are used to train, and the remaining subset is used to test. This process is performed k times until the whole dataset is tested [Alpaydin, 2014].

Each prediction can be classified as: i) *TP*: a vulnerable file correctly classified; ii) *FP*: a non-vulnerable file classified as vulnerable; iii) *TN*: a non-vulnerable file correctly classified; iv) and *FN*: a vulnerable file classified as non-vulnerable. The evaluation of each instance (experiment with a dataset configuration and an algorithm configuration) is done through the analysis of the precision, recall, and F-measure metrics. For the multiclass problem (classification per vulnerability category), each metric is calculated for each class.

6.1.2 Results and Discussion

This section presents and discusses the results obtained with the classical ML algorithms. We start with the binary classification results, then the results of the binary classification focusing on each category at a time, and conclude with the results of the multiclass classification (each class is a vulnerability category).

Table 6.3: Performance metrics for the binary classification (P : best precision, R : best recall)

Dataset Configuration	Precision	Recall	F-Measure	Algorithm
(P): SM + SAT	0.8855	0.0833	0.1522	Bagging
(R): SM + SAT	0.2328	0.9019	0.3701	Bagging
(P): SM	0.9404	0.0622	0.1166	Bagging
(R): SM	0.2274	0.8990	0.3630	XGB
(P): SAT	0.9389	0.0091	0.0181	Bagging
(R): SAT	0.1186	0.5835	0.1971	Bagging

Binary Classification

Table 6.3 shows the best performance results for the first experiment. As mentioned before, three dataset configurations are considered regarding the used features: *a*) SAT alerts and SMs, *b*) only SMs, and *c*) only SAT alerts. For each, we present only the ML algorithm that led to the best model in terms of precision (represented by the letter (P)) and the one that led to the best recall (represented by the letter (R)). As we can see, most are based on the *Bagging* algorithm, except one (best recall with only SMs as dataset configuration - *b*) that uses *XGB*. As for the remaining ML algorithms, they all presented worst results in all cases (their analysis is not included here as we prefer focusing on the impact of the dataset configurations/features).

The best precision is in the dataset with only SMs as features (0.9404). Although the best model for the dataset configuration with only SAT alerts has a similar precision (0.9389), the recall is much worse (0.0091 for SAT features and 0.0622 for SM features). This means that most of the files predicted as vulnerable are actually vulnerable files, at the cost of leaving most of the vulnerable files unidentified. Note that different cases in Table 6.2 are based on the same ML algorithm but use different hyperparameters (*e.g.*, for the dataset configuration using SAT alerts and SMs, the *bagging* algorithm is the best in both cases, but the one with the best precision does not use a sampling technique, while the one with the best recall uses random undersampling).

Regarding the dataset with SAT alerts and SMs, it has the worst precision among the three dataset configurations (0.8855). However, it achieves the best recall (0.0833) and the best f-measure (0.1522). This means that fewer non-vulnerable samples are misclassified at the cost of missing many vulnerable samples.

When considering the cases with the best recall, similar results are obtained for the dataset configurations with SMs and SAT alerts and the one with only SMs. The former has a recall of 0.9019 and a precision of 0.2328, while the latter has a recall of 0.8990 and a precision of 0.2274. The dataset with only SAT alerts has the worst results, with a recall of 0.5835 and a precision of 0.1186, meaning that most vulnerable samples are identified although there is a high number of FPs.

Table 6.4: Performance metrics for the binary classification per category - Memory Management (*P*: best precision, *R*: best recall)

Dataset Configuration	Memory Management			
	Precision	Recall	F-Measure	Algorithm
(P): SM + SAT	0.7996	0.1297	0.2232	Bagging
(R): SM + SAT	0.0490	0.8823	0.0928	XGB
(P): SM	0.8447	0.1479	0.2518	Bagging
(R): SM	0.0480	0.8826	0.0911	XGB
(P): SAT	0.5769	0.0597	0.1082	XGB
(R): SAT	0.0150	0.7028	0.0293	XGB

These results answer **RQ1**, as it is clear that SAT alerts cannot be used to predict the vulnerable files for the Mozilla dataset. This also helps answer **RQ2** since there are no promising results for the dataset with SAT alerts and SMs.

Binary Classification per Category

In this second experiment, we consider the three categories with the largest number of vulnerable files: *memory management*, *input validation*, and *permission*. Again, the letters (*P*) and (*R*) indicate the model with the best precision and the one with the best recall, respectively. As in the previous section, the algorithms with the best results are *bagging* and *XGB*.

Table 6.4 shows the results for the *memory management* category. The best precision is 0.8447 in the dataset configuration with only SM features. This value is worse than any of the best precision of the binary experiment (see Table 6.3). Regarding recall, slightly better results were achieved for the SM dataset (0.8826) than for the one with both SMs and SAT alert features (0.8823). Again, the worst results are obtained for the dataset with only SAT alert features. On the other hand, the best recall using this dataset configuration (*c*) is obtained in this category (0.7028).

The results for the *input validation* category can be seen in Table 6.5. The best precision is obtained in the dataset with both SMs and SAT alerts, with a value of 1, which means that all files predicted as vulnerable are actual vulnerable files. However, it comes at the cost of a very low recall (0.0391), meaning that most vulnerable files are left out of the prediction. Unlike what was noticed in the *memory management* category, this dataset allows achieving slightly better precision than the one with only SMs (0.9877). For both dataset configurations, there is a case with a recall above 0.88. Again, the dataset with only SAT alerts is the one with the worst results (precision 0.4072; recall 0.6556).

The overall conclusions for the *permission* category are similar to the ones for the *memory management* and *input validation* categories. The results can be seen

Table 6.5: Performance metrics for the binary classification per category - Input Validation (*P*: best precision, *R*: best recall)

Dataset Configuration	Input Validation			
	Precision	Recall	F-Measure	Algorithm
(P): SM + SAT	1.0000	0.0391	0.0753	Bagging
(R): SM + SAT	0.0365	0.8822	0.0702	Bagging
(P): SM	0.9877	0.0401	0.0771	Bagging
(R): SM	0.0363	0.8852	0.0696	Bagging
(P): SAT	0.4072	0.1022	0.1635	Bagging
(R): SAT	0.0126	0.6556	0.0246	XGB

Table 6.6: Performance metrics for the binary classification per category - Permission (*P*: best precision, *R*: best recall)

Dataset Configuration	Permission			
	Precision	Recall	F-Measure	Algorithm
(P): SM + SAT	0.8571	0.0143	0.0282	Bagging
(R): SM + SAT	0.0213	0.8948	0.0415	Bagging
(P): SM	0.8889	0.0064	0.0126	Bagging
(R): SM	0.0214	0.8956	0.0418	Bagging
(P): SAT	0.4124	0.0845	0.1402	Bagging
(R): SAT	0.0063	0.5625	0.0126	XGB

in Table 6.6. The best precision is 0.8889 in the dataset configuration with only SMs, which is worse than the results for the binary experiment with the same dataset configuration. With regards to recall, the best result is 0.8956, also in a configuration of the dataset with SMs as features. As for the other two cases analyzed above, the ML algorithms do not obtain good results when only the SAT alerts are used as features.

Based on the results above, we can say that the ML algorithms do not perform better at identifying vulnerable files per vulnerability category than in the binary experiment. This clearly answers **RQ3** about the ability of the ML algorithms to classify vulnerabilities per category. Nevertheless, it is important to highlight that the number of vulnerable instances for each vulnerability category is much smaller than in the dataset with all vulnerable files. Even for the largest category (*memory management*), the number of vulnerable files is only about 20% of all vulnerable files. The results also corroborate the idea that SAT alerts cannot be used as features to predict vulnerable files with ML algorithms (**RQ1**). Similarly, SAT alerts do not improve the prediction when using a SM dataset (**RQ2**) for the binary problem per category.

Table 6.7: Performance metrics per category (multiclass classification)

Class (Category)	Precision	Recall	F-Measure
Non-Vulnerable	0.9824	0.9999	0.9824
No Category	0.7692	0.6202	0.4394
Memory Management	0.6050	0.3419	0.3734
Input Validation	0.6602	0.2867	0.3364
Permission	0.5576	0.3386	0.3809
Cryptography	0.0007	0.0003	0.0004
Data Protection	0.3878	0.1218	0.1833
System Configuration	0.8347	0.5737	0.6508
File Management	0.0030	0.0024	0.0027
Coding Practices	0.6136	0.2547	0.3307

Multiclass Classification

Unlike the previous experiments, this one has more than two classes to be predicted by the ML algorithms. Hence, the performance metrics (precision, recall, f-measure) for these multiclass experiments are calculated per class [Ferri et al., 2009], as can be seen in Table 6.7. Note that the values in each category may not be achieved by the same ML model. Instead, we show the best values of each metric per category (*e.g.*, the best precision for the *memory management* category has been obtained with the *Bagging* algorithm and the best recall with XGB). This allows us to evaluate how well the ML algorithms can predict the vulnerable files in a multiclass configuration compared to the binary class configuration.

Each vulnerability category is represented as a class in the dataset, but some of the categories have few samples in the dataset (*e.g.*, *coding practices*, with 12 samples). A possible approach would be to keep only the largest categories and group the smallest ones as one class. However, we decided to keep them in the dataset as a separate class to evaluate if they can be detected.

Overall, the performance metrics of the multiclass experiment are worse than the ones of the binary class configurations. For example, considering the *memory management* category, the best precision for any model is 0.6050, which is smaller than the results of the binary prediction per category, regardless of the dataset configuration (the best result is 0.8447). The same happens for recall (0.3419 in the multiclass; 0.8826 in the binary). Similar observations can be done for the *input validation* (precision 0.6602; recall 0.2867) and *permission* (precision 0.5576; recall 0.3386) categories.

These results help answering **RQ3**, regarding the ML models ability to identify vulnerabilities per category. In fact, the results of the binary classification experiment are better than the ones obtained when categories are considered (both

binary per category and multiclass). Hence, we can conclude that it is possible to achieve slightly better results using binary classes (vulnerable/neutral) for this Mozilla dataset. This is probably because the number of samples in the binary classes is larger. Consequently, the ML algorithms have more information to create a model that better predicts vulnerabilities.

There are two categories (*cryptography* and *file management*) for which the performance metrics are almost zero. Together with vulnerabilities related to *coding practices* (last row of Table 6.7), these classes have the smallest number of samples in the dataset. This means that the algorithms probably cannot distinguish them during the training phase, which happens not only for the ML models presented in Table 6.7 but for several ML other models that achieve similar results. This also suggests that vulnerabilities in the *coding practices* category (precision 0.6136, recall 0.2547) can be better identified using ML algorithms with SAT alerts and SMs than the other categories (we observed good detection results with a reduced number of samples in this category). However, due to the reduced number of samples of vulnerable files in these three categories, this cannot be confirmed.

6.1.3 Understanding the Classification Results

As none of the ML models developed in the experiments presented before provide acceptable performance, we decided to investigate the characteristics of the source code files that led to these results. The goal is to study if there are any characteristics that may impact the predictions of the models. In fact, although we know that it is not possible to achieve perfect results without overfitting the data, we expected better results than the ones we obtained.

To try to understand why the performance is so low, we compared the SMs of vulnerable files with the corresponding neutral (non-vulnerable) versions. We focused on SMs as these led to better results in the experiments presented before. Also, SMs are a better indicator of the specific code characteristics than SAT alerts. The hypothesis is that the SM values of a vulnerable file are equal or similar to the ones of the neutral files. This way, we started by comparing all the vulnerable samples with the corresponding non-vulnerable samples in an attempt to find combinations with the exact same values for the 54 SMs considered.

Out of the 2,961 vulnerable files in the *dataset slice*, only 24 (0.81%) of them have at least a neutral version (of the same file) with exactly the same values for all SMs. Furthermore, for 45 vulnerable files, we could find at least one other file (not another version of the same file) with the same SM values. This is a small portion of the dataset samples and does not justify the poor results observed. This way, we decided to study the proportion of SMs with the same values in the vulnerable and non-vulnerable samples. The results are presented in the histogram in Figure 6.2, where the *x-axis* represents the number of metrics and the *y-axis* is the number of files. For example, the first column indicates that 150 vulnerable files have between 11 to 16 SM values that are equal to another non-vulnerable file in the dataset.

The histogram shows that, for almost half of the vulnerable samples (1,447 of

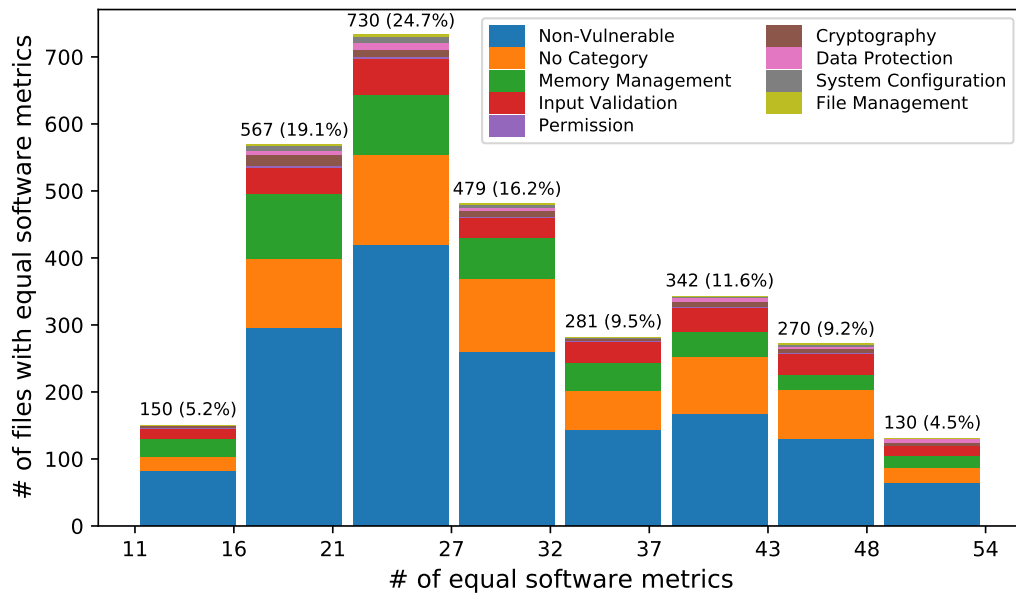


Figure 6.2: Histogram with the maximum number of equal software metrics in the vulnerable files compared with the neutral (non-vulnerable) files

the 2,961), less than 50% of the SMs have values that are equal to non-vulnerable samples. This can be seen in the three first columns of the histogram, which represent the files with 27 of the 54 SMs with equal values. Also, the largest number of shared SMs values is in the 21 to 27 interval, with 730 files (24.7% of the dataset). Moreover, the right-hand side of the histogram shows that only 130 vulnerable samples (4.4% of the dataset) have more than 48 SMs with equal values to at least one non-vulnerable sample.

The number of vulnerable samples having almost all SMs with the same value as neutral samples is not high (only 130 vulnerable samples if we consider the ones with more than 48 SMs). However, as shown in Figure 6.2, more than 50% of the vulnerable files have SM values similar to other neutral files. These characteristics make it hard to train ML models with good performance (as vulnerable and neutral files share many SM values, this makes it difficult for algorithms to distinguish vulnerable from neutral files).

Key Observations:

- The combination of SAT alert with SMs does not produce better results than using only SMs as features to predict software vulnerabilities using ML algorithms
- Vulnerable and neutral files share similar characteristics (a large number of SMs have the same values)
- The ensemble algorithms based on DTs perform better than others

6.2 Deep Learning

Yan et al. [2019] proposed MAGIC, which uses a deep learning technique to detect and categorize malware programs. Their proposal is divided into two steps. In the first one, they create and extract features from the CFG, which are used as input for the DGCNN network to create an embedding. In the second step, this embedding information is passed to an Artificial Neural Network inspired by VGG [Simonyan and Zisserman, 2014] to perform the classification, *i.e.*, to see if that program is malware or not.

Inspired by the results presented in [Yan et al., 2019], we adapted the approach to detect vulnerable C functions. In the same way MAGIC can be used to separate benign from non-benign (malware) programs, we hypothesized that it could be used to distinguish neutral (benign) from vulnerable functions (non-benign). In concrete, our research question (RQ4) is the following:

- **RQ4:** Can the MAGIC approach, based on DGCNN and VGG, detect vulnerable C functions using their CFGs?

In practice, the main contributions of this study are: *i)* the definition of memory management-related features for C functions, and *ii)* an evaluation of vulnerable C functions detection using the CFGs as input for DGCNN and VGG. During the evaluation, we managed to obtain high values for recall (above 0.96). However, the precision was very low, with values smaller than 0.04. Hence, we conclude that the technique does not provide results in detecting vulnerable functions as good as the ones obtained when classifying malware. In the experiments conducted by Yan et al. [2019] for malware detection, the dataset used was mostly balanced in what concerns the number of examples. However, our *dataset slice* has a small number of vulnerable functions (< 1%), which raises an additional challenge. Nevertheless, our results show that the models can successfully find most of the positive instances, *i.e.*, the vulnerable functions.

The remaining of this section is organized as follows. Section 6.2.1 presents the approach followed in the study, while Section 6.2.2 presents and discusses the results.

6.2.1 Vulnerabilities and Approach

A *dataset slice* of C functions from the Linux Kernel project, labeled either as vulnerable or neutral (non-vulnerable), is used in this study. Using the source code of the functions obtained from GitHub, we extracted their CFGs with the Joern tool [Yamaguchi, 2014]. Similarly to Yan et al. [2019], we extracted features related to code sequence and vertex structure. However, we enhanced the data with features related to memory management, as the most frequent types of vulnerabilities in the dataset are precisely related to such features: *CWE-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer* [MITRE, 2006b], *CWE-20*

- *Improper Input Validation* [MITRE, 2006d], and *CWE-399 - Resource Management Errors*) [MITRE, 2006e].

In the following, we describe each of the steps of the approach followed in this study. First, we explain the process to obtain the CFGs and the adjustments we performed on them. Then, we detail the process to extract the features for each node of the function CFGs. Finally, we present the network used to classify the functions. Figure 6.3 shows the high-level approach.

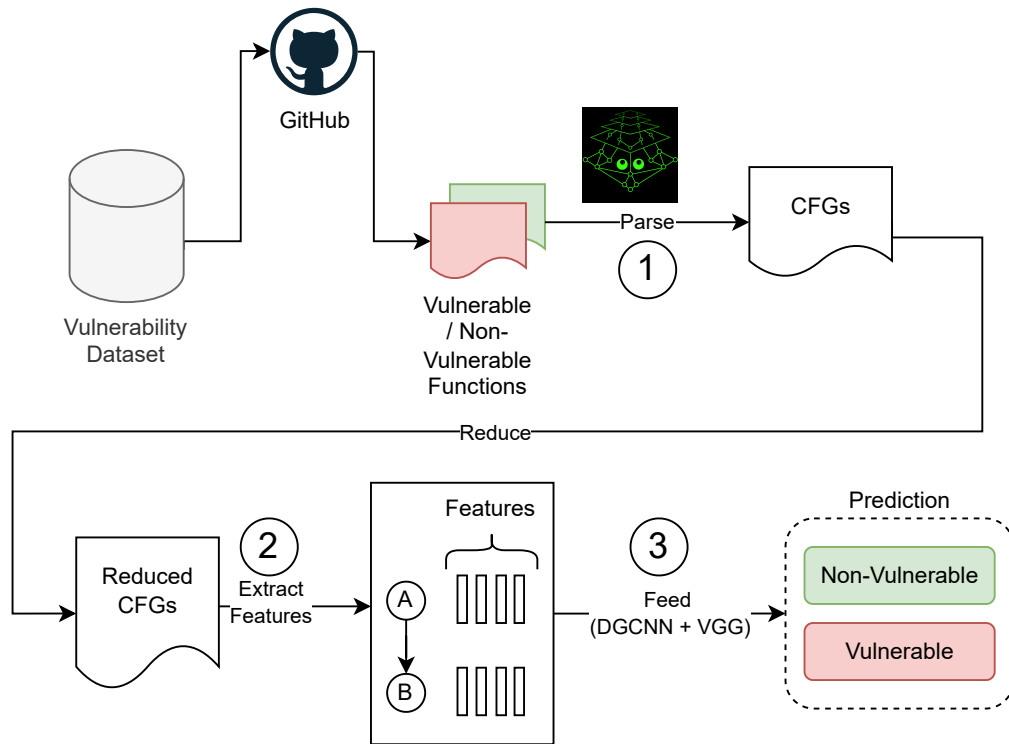


Figure 6.3: Approach followed to detect vulnerable functions

CFG Extraction and CFG Reduction

To obtain the CFGs, we used the Python application Joern [Yamaguchi, 2014], which was developed by Yamaguchi et al. [2014]. Although this application also provides other types of graphs related to the source code (*e.g.*, the AST), we decided to use only the CFG, as in [Yan et al., 2019]. To avoid issues related to the extracted CFGs, we manually reviewed some CFGs for some functions.

Joern models each statement from the C/C++ code as a node in the CFG and also extracts the actual statement. However, most functions contain several statements in a row without branching. As one of the features from Yan et al. [2019] is the number of instructions in a node, and to reduce the complexity of the problem, we decided to reduce the CFGs. Every time two or more nodes are sequential, we merge them into a single node. This can be done for all the nodes with both in-degree and out-degree equal to one (*i.e.*, only one edge reaching the node, and only one edge leaving the node) that have a previous node with an out-degree equals to zero or one (*i.e.*, only one edge leaving the previous node or

no edges leaving the previous node). An example of CFG reduction can be seen in Figure 6.4. Functions with degree one, *i.e.*, with only one node in the CFG, were removed from the classification process as their features would not help the training of the model to classify the functions.

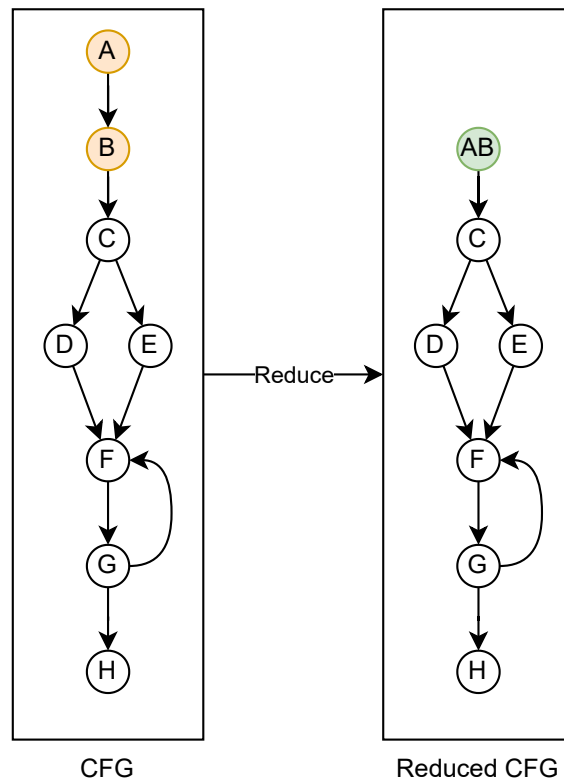


Figure 6.4: An example of a CFG being reduced. In this case, nodes A and B (in orange) are merged into a single node (in green)

Feature Extraction

The features extracted can be divided into three groups: *i)* Vertex Structure, *ii)* Code Sequence, and *iii)* Memory Management Features. Both Vertex Structure and Code Sequence features are adapted from the work of [Yan et al., 2019]. The complete list of features can be seen in Table 6.8.

Regarding the Vertex Structure features, we use the in-degree, the out-degree, and the number of instructions of a node. The in-degree is the count of edges that reach a certain node. Conversely, the out-degree is the count of edges that depart from a node. Differently from Yan et al. [2019], who use the degree of the node, we decided to use both the in-degree and out-degree as the CFGs are directed graphs. Additionally, we have the number of instructions that are in a node, which we could obtain due to the CFG reduction.

Regarding the code sequence features, we use the information obtained during the CFG extraction process. When the Joern tool extracts the CFGs, each node contains both the statement that generated it and its type. Examples of types include `and`, `greaterThan`, `assignment`, among others. All of these types were

Table 6.8: Features used for the DGCNN+VGG evaluation

Attribute Type	Attribute Description
From Vertex Structure	In-degree, Out-degree, # of instructions
From Code Sequence	transfer instructions, call instructions, arithmetic instructions, compare instructions, mov instructions, termination instructions, data declaration instructions
Related to Memory Management	allocation functions, deallocation functions, point assignment, memory address of, convert unsafe, string unsafe, scanf unsafe, and other unsafe functions

mapped into one of the code sequence features. The only feature not mapped here is “numeric constants”, as Joern did not report any node type that could be mapped into this feature. For example, the “assignment” type is mapped to the “mov instructions” feature.

We have created Memory Management features. The reasoning for using these lies in the fact that most of the vulnerabilities in the dataset are related to memory management issues. Hence, they may reveal important aspects of the vulnerability detection process. These features are:

- Allocation Functions: functions that allocate memory such as `malloc`, `kalloc`, `realloc`, `new`
- Deallocation Functions: functions that deallocate memory such as `free`, `delete`, `vfree`, `kfree`
- Point Assignment: all assignments that are made in a pointer
- Memory Address Of: all node types labeled with type `addressOf`
- Convert Unsafe: functions considered unsafe when converting the types of a variable, such as `atoi`, `atol`, `atoll`, and `atof`
- String Unsafe: the use of string manipulation functions that are considered unsafe, such as `gets`, `getpw`, `strcat`, `strcpy`, `sprintf`
- Scanf unsafe: the use of functions to scan streams of data such as `scanf`, `fscanf`, `sscanf`, `vscanf`;
- Other unsafe functions: the use of other unsafe functions such as `realpath`, `getopt`, `getpass`, `streadd`

Classification Framework

The features described in the previous section serve as input to an artificial neural network used to perform the classification. This follows the same methodology proposed by Yan et al. [2019], consisting of two networks: a DGCNN and a VGG.

The first part of the classification framework is a DGCNN, which is a CNN tailored for graphs. As with any deep learning classification model, we have a first layer where the input is the sample (in our case, the CFG of the function). Then, we have several hidden fully connected layers. Finally, we have an output layer, which can either produce the classification or features to be used by another model. We moved with the former approach, as we have another network to be fed (*i.e.*, VGG).

As the samples in our dataset (functions) are CFGs, they may have different sizes. Hence, we need a way to reduce them to the same size. Otherwise, the network will not be able to be trained and later perform the classification. Although the number of features is the same for each node of the CFG, the number of nodes changes, as each function may have a different number of statements. Hence, we use a DGCNN approach and perform a pooling mechanism in the last layer of the network.

Although Yan et al. [2019] considered two different pooling approaches (Sort-Pooling and AdaptiveMaxPooling), we decided to use the one that produces the best results: Adaptive Max Pooling (AMP). AMP [Yan et al., 2019] consists of pooling the features from a graph using a predefined output size. For instance, if we use a 3×3 AMP, all samples (CFGs in our case) will be reduced to an output of 3×3 , regardless of the number of nodes that each CFG has. To do that, the AMP algorithm adjusts both the kernel size (small matrix applied to the input to pool the max value of that small window) as well as the stride (the movement that the kernel does to pool the next value) according to the input to AMP algorithm.

AMP is applied after the last layer of DGCNN, and its output is used as input to VGG, which will perform the final prediction. In our case, we are using VGG11. This is the same approach followed by Yan et al. [2019] in MAGIC. Figure 6.5 shows the detailed approach used to classify the functions, while Figure 6.6 shows how AMP works. Table 6.9 shows the hyperparameters used for this evaluation (adopted from Yan et al. [2019]).

Even though Yan et al. [2019] do not explicitly use normalization, we decided to use a min-max normalization process, where all the features are normalized before being used by the DGCNN process. This was done to smooth the variation in the feature values (that, in our case, may have very different domains). No feature selection technique is used, as the number of features is small (*i.e.*, 18).

6.2.2 Results and Discussion

This section presents the results obtained during the evaluation of DGCNN + VGG approach to detect vulnerable C functions. We also discuss and cast some

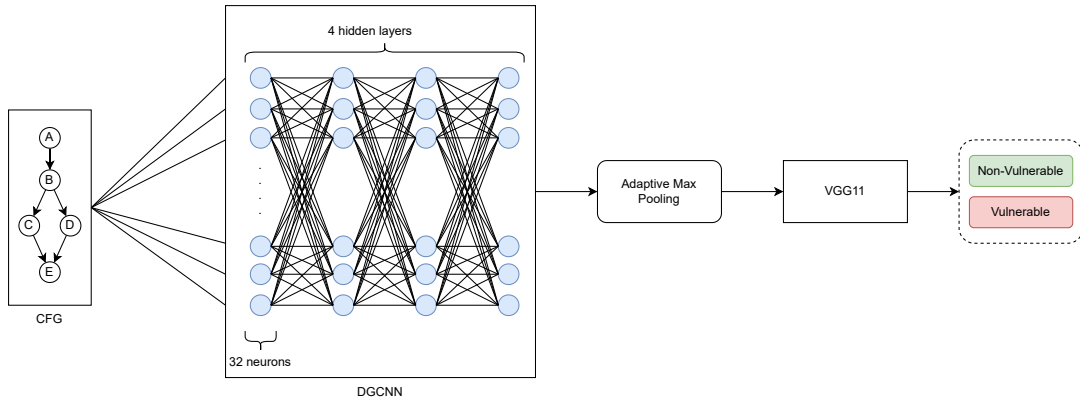


Figure 6.5: Detailed approach of the deep learning classification process

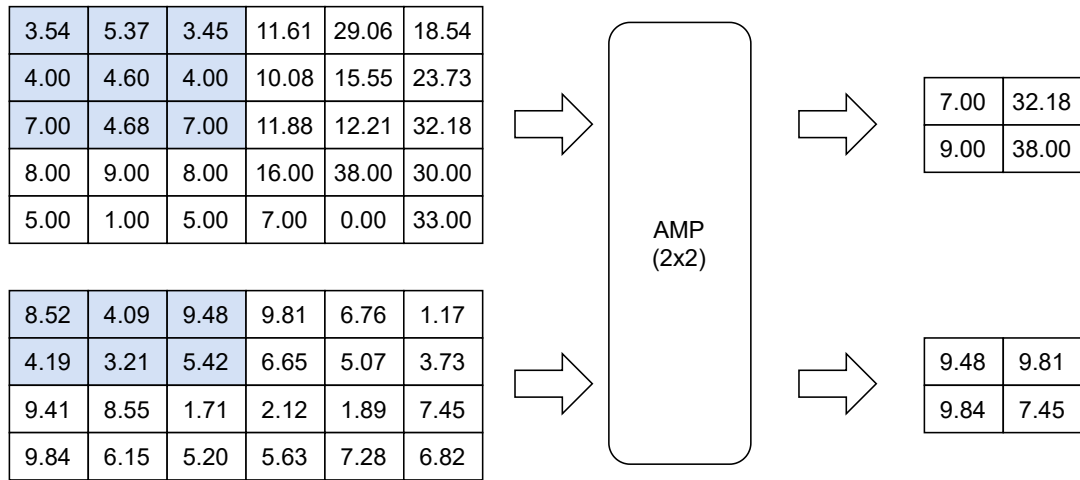


Figure 6.6: An example of a 2x2 Adaptive Max Pooling with different input sizes. The kernel size (highlighted window) is different in order to have the same output size. For both cases, the stride was 3x2, padding = 0 (adapted from [Yan et al., 2019])

Table 6.9: Algorithms and Techniques

Parameter	Values
Algorithm	DGCNN + VGG
Normalization	Min-Max
Pooling Type	Adaptive Max Pooling
Pooling Ratio	0.3
Graph Convolution Size	(32, 32, 32, 32)
2D Convolutional Channels	[16, 32]
Dropout Rate	[0.1, 0.5]
Batch Size	[10, 40]

light on the possible reasons concerning the obtained results. As before, we focus on precision, recall, and F-measure metrics.

The combination of the selected hyper-parameters generated different algorithmic configurations. Each configuration run takes from 17.5 to 18.5 hours to execute. Note that we performed preliminary experiments without the memory management-related features (*i.e.*, only with the vertex structure and code sequence features), but the results were not promising. Hence, we focused on the experiments with all features. Nevertheless, the results were pretty similar. Precision varied from 3.36% to 3.43%, recall varied from 96.93% to 99.38%, while F-measure varied from 6.50% to 6.63%. Looking at the recall results, it is possible to see that the model can successfully identify most of the vulnerable functions (recall > 98%). However, when we take a look at the precision values, we see that the model reports a large number of FPs (see Figure 6.7, as an example of a Confusion Matrix), *i.e.*, it is classifying non-vulnerable functions as vulnerable. Although this is preferable to classifying vulnerable functions as non-vulnerable, a very high number of FPs makes the results useless.

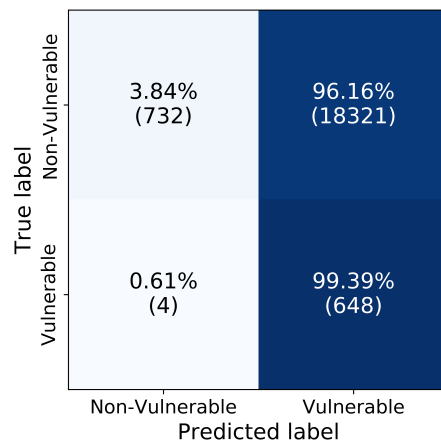


Figure 6.7: Confusion Matrix with the prediction for an experiment run

Compared to Yan et al. [2019], who obtained precision, recall, and F-measure with values higher than 0.96 for one of the datasets that they analyzed, our results are much worse. There are three main reasons for that.

The first is concerned with the number of examples of each type of function that we have to perform the training of the ML model. After removing functions with degree one, the *dataset slice* used includes 65,685 functions (samples), being 2,160 (3.29%) vulnerable, and the remaining 63,525 (96.71%) non-vulnerable. This is a highly unbalanced dataset. The datasets used by Yan et al. [2019] to benchmark the approach were not as severely unbalanced as ours. For the MSKCFG dataset, the largest class has 2,942 samples (27.07%) out of 10,868 samples, while the largest class for the YANCFG dataset has 3,980 samples (24.34%) out of 16,351 samples.

Second, the sizes of the CFGs are quite different. More than one-fourth of our dataset has functions with a degree of one. Though we removed them for the training process, the median of the degree is 10, and more than 75% of the functions lead to 19 nodes or less in the CFG. Conversely, CFGs in the work by Yan et al. [2019] had at least 30 nodes, being the median 241. Consequently, a larger

number of features were available to be selected by the pooling mechanism during the use of the AMP technique.

Finally, the nature of the problem is different. While Yan et al. [2019] aim to classify different classes of malware, for which almost all classes can cause damage to the end user, our goal is to separate the potentially problematic functions (vulnerable) from the neutral functions (non-vulnerable). Although the malware classes in [Yan et al., 2019] also contain a “benign” class, the number of samples in this class is small. Furthermore, the damage caused by a wrong classification, in our case, is much more severe than a wrong malware class classification.

Regarding the answer to **RQ4**, and from a technical point of view, it is possible to use the MAGIC framework, which relies on DGCNN + VGG, to detect vulnerable C functions based on their CFGs. However, the results obtained in our experiments, using features related to vertex structure, code sequence, and memory management, yielded poor results when distinguishing between vulnerable and neutral (non-vulnerable) functions in the Linux dataset due to the low number of existing vulnerabilities, which results in a highly unbalanced dataset. This opens new avenues for further research, namely regarding the development of more informative features that allow for a better distinction between vulnerable and non-vulnerable functions and/or the development of techniques to create a more balanced dataset.

Key Observations:

- A deep learning approach (based on DGCNN + VGG) can obtain high recall when predicting vulnerable functions at the cost of a low precision
- Compared to MAGIC, the dataset is highly imbalanced, which represents a key challenge

6.3 Threats to Validity

In this section, we discuss the limitations and threats to the validity of the two studies presented in this chapter.

External Validity refers to the ability to generalize the results beyond the experiment settings. The first threat concerns the number of projects used in the studies. The first study considered only Mozilla, while the second considered only the Linux Kernel. Although they are representative open-source projects, it is harder to generalize the obtained results in another context, and the study would be more complete if more projects were used. Other projects could have been used to generalize these findings. Nevertheless, the results presented are still relevant, as they are large projects with a wide variety of vulnerabilities over a large period.

The second threat is the SATs selected in the first study. In fact, the results of the ML algorithms are restricted to the quality of CppCheck and Flawfinder. Al-

though these SATs have limitations (as we noticed in Chapter 4), they are widely used in the industry [Arusoaie et al., 2017].

The third threat is related to the nature of the ML algorithms used in the first study, as they are all based on decision trees. RF, XGB, and Bagging are ensemble algorithms that use DT as weak learners to come up with a final prediction. The results could have been different if classification algorithms that use different techniques were used (for instance, SVM). To confirm this, we conducted a preliminary experiment with other algorithms (SVM and Neural Network (NN)), but the results were also poor (precision and recall below 0.70). It is thus clear that the information provided by SMs and SAT alerts is not enough to create good prediction models using ML algorithms.

Construct Validity refers to the number of vulnerable samples per category. This is the fourth threat and applies to the first study. In fact, even in the largest category (*memory management*), the number of vulnerable files is only 603, and it is expected that having more vulnerable samples would improve the performance of the ML algorithms. Although sampling techniques were used to mitigate this issue, this was not enough to achieve better performance metrics.

The next threat is related to the maturity of the selected projects. Both Mozilla and Linux Kernel are stable projects, and although they are open-source projects, they follow a very mature development process. Consequently, in the second study, the functions may be very similar, resulting in similar CFGs, regardless of whether they are vulnerable or not.

Internal Validity refers to the possibility of having unanticipated relationships. This threat is related to the application used to extract the CFGs (Joern [Yamaguchi, 2014]) in the second study. Although it is a widely used application, it may have defects. Nevertheless, we used it to extract the CFGs for our study and manually reviewed some of the functions with the extracted CFGs to guarantee that we would not find further issues.

6.4 Summary

This chapter studied the use of ML algorithms (either classical ones or using deep learning) to detect vulnerable code units in C/C++ open-source projects. In the first study, three sets of features were used: SAT alerts, SMs, and both SAT alerts and SMs. Experiments using all vulnerabilities and vulnerabilities classified per category were performed. The results show that combining SMs with SAT alerts does not significantly change the vulnerability prediction when compared with using SMs and SATs alerts separately. Additionally, a manual analysis showed that vulnerable and neutral (non-vulnerable) files share similar characteristics, making the vulnerability prediction harder using ML on top of SMs and SAT alerts. Anyway, the ML algorithms that perform better are ensemble algorithms based on DTs. They can achieve good precision or recall, but not both at the same time.

In the second study, we evaluated the possibility of using a state-of-the-art deep learning technique (with DGCNN and VGG) to classify vulnerable C functions of the open-source project Linux Kernel. Our approach is based on the work developed by Yan et al. [2019] and uses the CFGs of the functions to collect features that an Artificial Neural Network then uses to detect if a function is vulnerable or not. In concrete, we used the Joern tool to extract the CFGs and reduced them to decrease the training time. In addition to the features related to vertex structure and code sequence, we added new types of features related to memory management. The results show that the proposed approach can successfully identify the vulnerable functions and obtain high recall values (greater than 0.96). However, the precision values are very low, indicating that the framework classifies non-vulnerable (neutral) functions as vulnerable. Even though this is the preferable scenario, it is not ideal. The reason for these results might be explained by the fact that we have a highly unbalanced dataset, where few examples correspond to vulnerable functions.

Based on the results of the two studies, we can conclude that none of the ML approaches considered is able to predict software vulnerabilities in an acceptable manner. Although we can obtain good precision and good recall in some cases, no configuration of the hyperparameters allowed us to obtain both simultaneously. Consequently, either a high number of FPs or FNs is obtained. For this reason, we decided to move forward and propose an approach that, instead of attempting to classify code units as vulnerable or not, provides an overall characterization of those units, considering their proneness to be vulnerable. Such characterization can guide the development team when addressing potential security hotspots. This approach, named Security Characterization of Open-source functions using Logic Scoring of Preference (SCOLP), is presented in the next chapter.

Chapter 7

Characterizing Code Units with Logic Scoring of Preference

SCOLP (Security Characterization of Open-source function using Logic Scoring of Preference) is a methodology that, based on static properties of the source code, is able to categorize code units (*e.g.*, functions) into priority groups considering the perceived proneness to have security issues. SCOLP is based on Logic Scoring of Preference (LSP) [Dujmovic, 2018], a well-known MCDM technique. In practice, characteristics extracted from the source code (*e.g.*, SMs and the attributes related to memory management, as introduced in the previous chapter) are used as input for LSP that, in the present work, is implemented through Quality Models (QMs). A QM is a tree structure composed of attributes, weights, thresholds, and operators that are combined to calculate a score for each entry (*i.e.*, each code function under assessment).

Due to the complexity of the problem, we focus on C functions from large open-source software projects. This way, we developed a set of QMs that take SMs and memory management-related attributes and output a score to support the categorization. Each QM emphasizes different characteristics of the code by assigning different weights to the input attributes and using diverse aggregation operators. We use SMs as these portray quality characteristics that are usually associated with the existence of bugs (volume and complexity) [Meiliana et al., 2017; Premraj and Herzig, 2011], and memory management-related attributes as these are related to very frequent vulnerabilities in C code. Based on the output of the QMs, code units are categorized into five priority groups that, in practice, provide a prioritization of the code units from a vulnerability susceptibility perspective: critical, high, medium, low, and lowest. Note that by integrating our mechanism into a CI pipeline, developers can be continuously informed when functions do not achieve a minimum categorization level.

We demonstrate the proposed solution on top of Linux Kernel, one of the projects from the dataset presented in Chapter 3. As expected, different categorizations are provided by the different QMs (allowing teams to define their own priorities), but, in most cases, the vulnerable versions are placed in the most critical priority groups, while the non-vulnerable (neutral) ones are placed in the less critical

groups. It is important to emphasize that the goal of Security Characterization of Open-source functions using Logic Scoring of Preference (SCOLP) is not to detect vulnerabilities or vulnerable code units (like the studies of the previous chapter) but instead to categorize code based on the perceived propensity to be vulnerable, which our results show to be the case.

To validate SCOLP, we compare the categorizations with the view of a group of security experts. A small set of vulnerable and non-vulnerable functions was chosen to be manually categorized by the experts into the priority groups mentioned above, and the outcome was later compared with SCOLP mapping. In practice, even considering the subjectiveness of individual judgments, we observe that the SCOLP categorization is aligned with the categorizations provided by the security experts (the functions were mapped into the same or a very close priority group in most cases).

The remainder of this chapter is organized as follows. Section 7.1 presents SCOLP. The different QM instances are discussed in Section 7.2. Section 7.3 presents the experimental evaluation and discusses the results. The validation based on the judgment of security experts is in Section 7.4. Section 7.5 discusses the threats to the validity. Finally, Section 7.6 summarizes the chapter.

7.1 SCOLP Methodology

SCOLP is based on Logic Scoring of Preference (LSP), a Multi-Criteria Decision Making (MCDM) approach that allows considering specific preferences in the categorization process. In practice, LSP allows the evaluation of complex systems to be made by comparing the attributes of the evaluated object (in our case a code unit) against the predefined criteria [Dujmovic, 2018] (which is not the case for other techniques such as some ML algorithms).

Figure 7.1 depicts the four steps of SCOLP to characterize code units from a security perspective. The process starts with the definition of the QMs, followed by the extraction of the attributes (*e.g.*, SMs) to feed the QMs and the score computation. This score is then used to map the code units into the priority groups. As our approach allows using several QMs simultaneously, the final step combines the different categorizations. This is an optional step, which is only required if more than one QM is being used and the software development team wants to get a combined categorization (instead of the individual ones). These steps are detailed in the next sections.

7.1.1 Quality Model Definition

The first step is to define one or more QMs based on static attributes of the source code. A QM is a MCDM technique based on LSP used to evaluate the quality of products, services, or processes. In our case, the QM outputs a score that is used to assign the code units (*e.g.*, functions) to a priority group. To create a QM, which has a tree structure, the following properties need to be defined: *i*) Attributes,

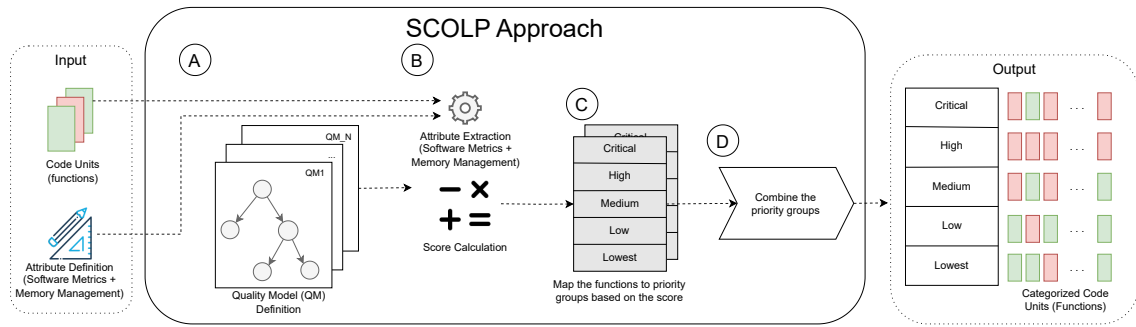


Figure 7.1: SCOLP approach followed to characterize the code units (functions)

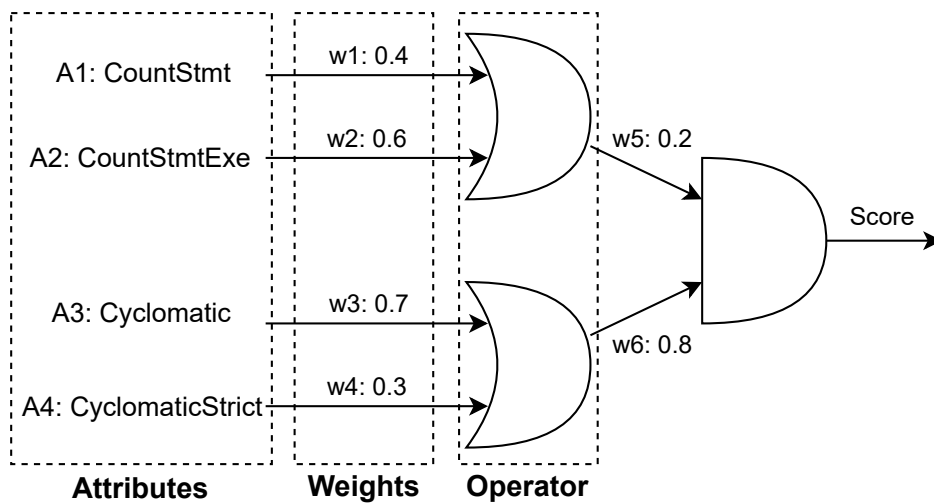


Figure 7.2: Example of a quality model

ii) Weights, iii) Thresholds, and iv) Operators. Figure 7.2 shows an illustrative example of a QM. This QM has four attributes (*CountStmt*, *CountStmtExe*, *Cyclomatic*, and *CyclomaticStrict*), each of them with an assigned weight (w_1 , w_2 , w_3 , and w_4 , respectively). They are combined with two operators that output two values. Each value has another weight (w_5 and w_6) to be combined by the final operator. As a result, a score for the QM is obtained.

The interpretation of the score depends on the type of attributes, which can be either a benefit or a cost attribute. In a benefit attribute, the lower values represent the worst values and the higher values represent the best values. An example of a benefit attribute is throughput. For cost attributes, the interpretation is the opposite, and the lower the values, the better. For instance, memory usage is a cost-type attribute.

To create a QM we need to define the QM focus, which will guide the selection of the *attributes*. Attributes are the quantification of a characteristic of the target object of the QM (*i.e.*, the code units in our case). Different QMs can target specific security aspects, such as *memory management*, *input validation*, and *permission* (see vulnerability categories defined in Chapter 3). For example, if we want to evaluate *memory management* characteristics, we can use the number of memory allocation functions in the source code as an attribute.

For each attribute, we need to define the *thresholds*, specifying the minimum and maximum acceptable values. If an attribute has a value that is outside the range of minimum and maximum acceptable values, the value considered for the score calculation should be the one closest to the threshold value for that attribute. For instance, if an attribute accepts values between 1 and 50, and a code unit has a value 52 for that attribute, then 50 should be considered for the calculation.

In some cases, attribute values may need to be normalized to use a common scale, such as from 0 to 1. The first alternative is *min-max normalization*, using the following equation:

$$s(x) = \frac{x - X_{min}}{X_{max} - X_{min}}$$

where $s(x)$ is the scaled value for an attribute whose value is x . X_{min} and X_{max} represent the minimum and maximum value allowed for that attribute, respectively. The second normalization approach is *Z-Score*, using the following equation:

$$s(x) = \frac{x - \mu}{\sigma}$$

where μ indicates the average of the attribute, and σ the standard deviation for the attribute.

Weights need to be assigned to each QM tree node. The leaf nodes are the attributes, and the others are the aggregation of leaf nodes or other aggregation nodes. *Dujmovic* explains seven ways to assign the weights to each node [Dujmovic, 2018]. Regardless of the technique, the sum of all weights of the attributes under a node should be 1. We use three of those techniques, which help to understand the impact on the aggregated scores and can be easily applied in SCOLP (find further details in [Dujmovic, 2018]):

- **Importance Decomposition Method (IDM):** each attribute weight is defined based on the overall qualitative importance among all the attributes of the upper node. The importance helps define other variables according to the aggregation mechanism. Table 7.1 shows the range of importance levels possible and their *andness* propensity.
- **Direct Weight Assessment (DWA):** this is the most straightforward technique, in which a person defines each attribute weight based on their individual judgment.
- **Weight based on Ranking (WBR):** in this technique, each attribute is ranked from 1 to n , where n is the number of attributes that form the upper node. The most important attribute is ranked 1, the second most important is ranked 2, and so on until the least important one, which is ranked n . After that, each weight W_i is defined using its importance (ranking) and the total number of attributes with the following formula:

Table 7.1: Attributes overall importance (adapted from [Dujmovic, 2018])

α	S	Overall Importance
1.0000	16	Highest
0.9375	15	Slightly below highest
0.8750	14	Very high
0.8125	13	Slightly above high
0.7500	12	High
0.6875	11	Slightly below high
0.6250	10	Medium-high
0.5625	9	Slightly above medium
0.5000	8	Medium
...

$$W_i = \frac{2(n + 1 - i)}{n(n + 1)}$$

The *operators* define how each attribute should be used in the aggregation. They can be any of the following:

- **Simultaneity:** when two or more attributes should be satisfied. This is equivalent to the *and* logic operator, and it is called full conjunction
- **Substitutability:** when either of the attributes should be satisfied. This is equivalent to the *or* logic operator, and it is called full disjunction
- **Neutrality:** is the arithmetic mean of both simultaneity and substitutability

There are some rules for the use of the operators, such as in the case of full conjunction (logic *and*), where the aggregated value can be at most equal to the minimum value of all the input attributes (*e.g.*, when aggregating 0.3 and 0.1 the result is 0.1). This allows, for example, to consider several volume metrics in the evaluation but allows the criteria to be equally satisfied with any of them. To provide flexibility regarding how each attribute contributes to the score, other operators ranging from full conjunction (simultaneity) to full disjunction (substitutability) are defined in [Dujmovic, 2018] and summarized in Table 7.2.

There are several techniques to aggregate the attribute values and calculate the score [Dujmovic, 2018]. In SCOLP, we use two of them as they can be easily applied in our context:

- (i) **Simple Arithmetic Mean:** the defined weights are used to aggregate the scores directly

Table 7.2: GGCD.17 operators (adapted from [Dujmovic, 2018])

Type of logic model	Form	Mode	Level	Symbol	Andness	Orness
Substitutability	Pure disjunction	Hard	Extreme	D	0.0000	1.0000
			Very high	D++	0.0625	0.9375
	Partial disjunction	Hard	High	D+	0.1250	0.8750
			Mid-high	D+-	0.1875	0.8125
			Medium	DA	0.2500	0.7500
		Soft	Mid-low	D-+	0.3125	0.6875
			Low	D-	0.3750	0.6250
			Very low	D-	0.4375	0.5625
Neutrality			A	0.5000	0.5000	
Simultaneity	Partial conjunction	Soft	Very low	C-	0.5625	0.4375
			Low	C-	0.6250	0.3750
			Mid-low	C-+	0.6875	0.3125
		Hard	Medium	CA	0.7500	0.2500
			Med-high	C+-	0.8125	0.1875
			Very high	C++	0.9375	0.0625
	Pure conjunction	Hard	Extreme	C	1.0000	0.0000

- (ii) **Weighted Power Mean:** not only the weights are used to calculate the score, but also the coefficient that represents the operator, as illustrated in Table 7.2

For the *Weighted Power Mean* case, in addition to pure disjunction (*or* operator), neutral, and pure conjunction (*and* operator), there are intermediate aggregation levels that can be used in a Generalized Conjunction/Disjunction (GCD). One example is GGCD.17, which can be seen in Table 7.2. The number in the name of the aggregation level represents the 17 possible values for the operators, varying from the pure disjunction to the pure conjunction.

7.1.2 Attribute Extraction and Score Calculation

The second step is to extract the attribute values of code units, which are used as the input for the QMs. In this work, we propose the use of static information that can be obtained from the source code (*i.e.*, by running SATs over the source code). These attributes were chosen as they can be easily extracted and integrated into a CI pipeline. Nevertheless, properties other than static could be used in SCOLP.

Using the QMs (attributes, thresholds, weights, and operators) and the attribute values, we can compute a score. In practice, each code unit gets a score used to categorize it. Note that scores represent the aggregation of several characteristics

based on some subjective criteria specified by human experts and, thus, do not have any concrete meaning.

7.1.3 Map Code Units into Priority Groups

Using the score for each code unit being assessed, we can map them into priority groups. We propose dividing each priority group boundary uniformly, considering the minimum and maximum values of the scores. The number of groups can be configured. For example, if we consider five priority groups (as in our experiments) and the score varies from 0.0 to 1.0, the first group has boundaries 0.0 and 0.2. The code units with scores greater than the minimum (0.0) and smaller than the maximum (0.2) are mapped into this group. Other approaches may be used for defining the boundaries of each priority group (*e.g.*, to allow the software development team to adjust the categorization according to the needs). We propose the following priority groups to be used in most cases:

- (i) *Critical*: the source code is hard to understand and very likely to be faulty or vulnerable (scores in range]0.8..1])
- (ii) *High*: the source code has some complexity, but it can be understood after reading it carefully, being still prone to be vulnerable (scores in range]0.6..0.8])
- (iii) *Medium*: the source code can be easily understood, but it would benefit from refactoring to reduce the probability of having vulnerabilities (scores in range]0.4..0.6])
- (iv) *Low*: the source code is easy to understand and not so prone to be vulnerable (scores in range]0.2..0.4])
- (v) *Lowest*: the source code does not need any maintenance change (scores in range [0.0..0.2])

The assignment to the priority groups may require require adjustments depending on the characteristics of the code units and the software development context. Such adjustments should be decided by the users of SCOLP. For example, in our experiences (see Section 7.3), we decided to perform some adjustments, namely shifting code units that do not have a minimum quality criteria (*e.g.*, outliers in terms of size) to the more critical priority groups.

7.1.4 Combine Priority Groups into a Unified Categorization

The final step is to combine the categorization output obtained by applying each QM into a single categorization. This is an optional step that can be used when the code units have special characteristics or when a software development team wants to consider specific types of vulnerabilities that require using alternative models. For instance, large functions may require a QM different from the one

used for small functions; or functions that essentially do memory management operations may require a QM different from those that mostly do database access operations. Different QMs should be used for different code units in this case. Thus, the output priority groups must be combined. For example, each QM can be assigned a weight, and the outputs can be combined using LSP through an aggregation technique such as simple arithmetic mean or weighted power mean.

7.2 Quality Model (QM) Instances

Our QMs are based on two sets of attributes that can be directly extracted from the source code: (i) SMs, and (ii) memory management-related attributes. While the former has been used in several contexts (e.g., detect faults [Meiliana et al., 2017; Premraj and Herzig, 2011], detect vulnerabilities [Walden et al., 2014]), the latter is usually related to most vulnerabilities in C language, as discussed in Chapter 5.

SMs can be extracted at different code levels (e.g., file, function, and class level), as presented in Chapter 3. However, only the function level SMs are used in this work, as these represent a fine-grained characterization of the code, allowing the software development team to be more effective in the analysis (less code is analyzed than if file or class SMs were considered). Nevertheless, metrics at other levels can also be used to build alternative QMs.

Regarding the memory management-related attributes, we extract the CFGs of each function and count the number of occurrences of the relevant types of nodes. For instance, we count all the memory allocation functions (such as `malloc`) and use that as the value for the *memory allocation functions* attribute. Obviously, the attributes should be related to relevant security issues. The attributes considered in this work (memory management-related) are the ones defined in the previous chapter (see Section 6.2.1).

To create the QM instances, we defined a baseline QM with SMs and used it to create other models focusing on different characteristics. As most vulnerabilities in C projects are related to improper memory management, we created specialized QMs that focus on these security characteristics, namely: *Memory Management* (MM), *Input Validation* (IV), and *Permission* (PER). These characteristics are indicative of the most frequent vulnerability categories in open-source C/C++ projects [Higgins, 2020; Team, 2022], namely in the Linux Kernel project used in our experimental evaluation. QMs including such characteristics allow SCOLP to provide priority group assignments with an enhanced security perspective.

The base QM, depicted in Figure 7.3, comprises three groups of attributes: two groups related to volume (statements and lines of code) and another group related to complexity. The metrics considered are some of the most used to characterize the size and complexity of the code. The operators are either variations of conjunction (*and*) or of disjunction (*or*), as discussed in the previous section and summarized in Table 7.2 [Dujmovic, 2018].

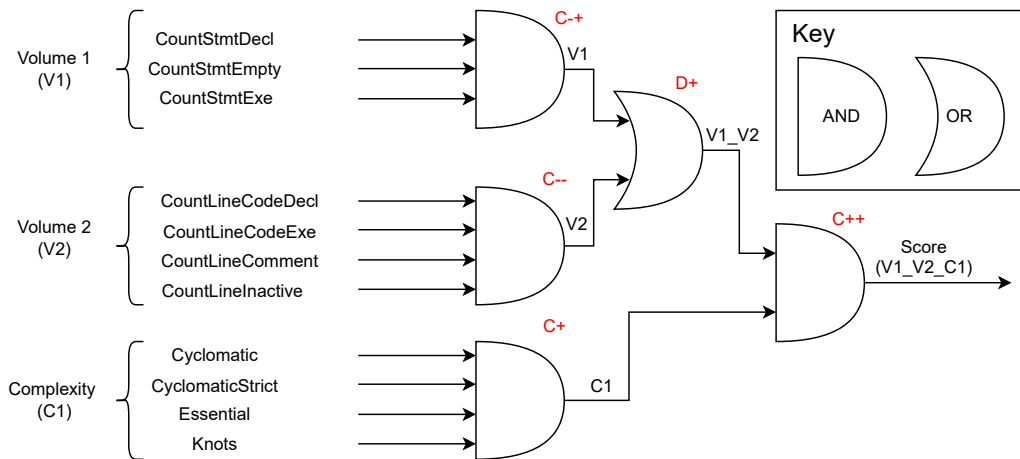


Figure 7.3: QM using Software Metrics of volume and complexity

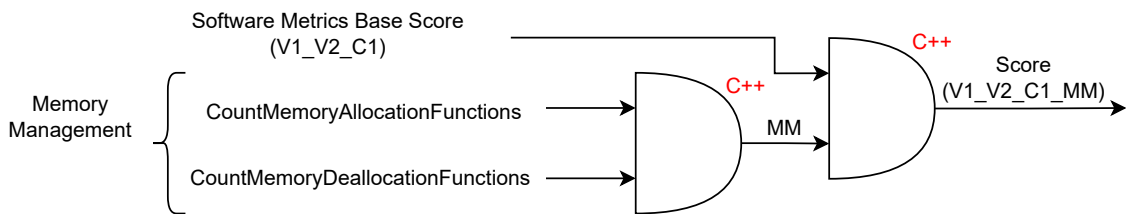


Figure 7.4: Quality Model with Memory Management (MM) attributes

The **first specialized QM** instance focuses on Memory Management (MM) and can be seen in Figure 7.4. As this model aims to identify potential security issues related to improper memory management, we added two attributes related to the use of functions that are well known to lead to such vulnerabilities: the number of memory allocation functions and the number of memory deallocation functions.

The **second specialized QM** focuses on Input Validation (IV) and can be seen in Figure 7.5. The attributes on this model are related to how the user input is handled and are divided into two groups: the first one has one attribute based on the count of address of statements (as the user input may influence the address location); while the second focuses on the occurrences of unsafe functions being used. Such functions can be related to data type conversion, string conversion, scanf (family of C functions that read the input from the standard input¹), or

¹ https://www.tutorialspoint.com/c_standard_library/c_function_scanf.htm

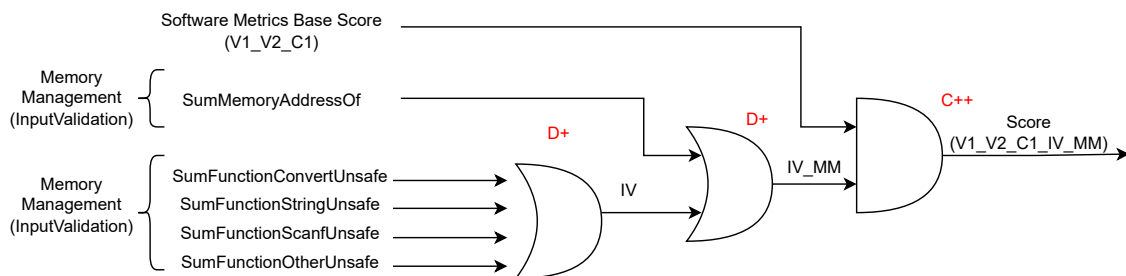


Figure 7.5: Quality Model with Input Validation (IV) attributes

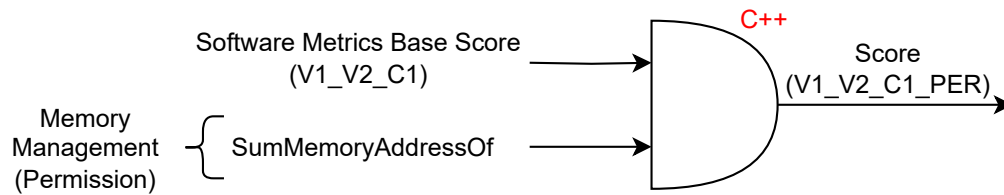


Figure 7.6: Quality Model with Permission (PER) attributes

Table 7.3: Configurations used in the experiments

Configuration	Possible Alternatives
Quality Model	QM of SMs, Memory Management QM, Input Validation QM, Permission QM
Label	Binary, Multiclass
Normalization Technique	Min-Max, Z-Score
Weight Assignment Technique	Importance Decomposition Method (IDM), Direct Weight Assessment (DWA), Weight based on Ranking (WBR)
Aggregation Mechanism	Simple arithmetic mean, weighted power mean

other unsafe uses. The outputs of these two groups are aggregated, and the result is combined with the base QM score.

The **last specialized QM** focuses on Permission (PER) aspects. This model aims to identify code units with potential permission violations. Only one additional attribute is considered (count of memory address of), representing the existence of code patterns that potentially violate some permission when accessing unauthorized memory areas.

In the experiments, we used several configurations: we varied the QMs, the weight assignment technique, and the aggregation mechanism. A summary of the configurations is in Table 7.3.

7.3 Experimental Evaluation

The following sections present the experimental evaluation of SCOLP. First, We introduce the vulnerabilities considered and the approach of the evaluation. Then, we present and discuss the results.

Table 7.4: Summary of the functions per vulnerability category (when vulnerable) of the Linux Kernel project

Category	CWEs	# of Functions (%)
Memory Management	119, 125*, 190*, 362, 399, 400*, 404*, 416, 476, 772*, 787*, 824	37 (7.30%)
Input Validation	20, 78, 79, 91, 94, 134, 189	2 (0.39%)
Permission	255, 264, 269, 284, 287, 352, 862*	1 (0.20%)
Data Protection	199, 200	2 (0.39%)
Coding Practices	17, 19, 254	0 (0.00%)
Cryptography	310	0 (0.00%)
System Configuration	16	0 (0.00%)
File Management	22, 59	0 (0.00%)
Output Encoding	-	0 (0.00%)
Error Handling and Logging	-	0 (0.00%)
Communication Security	-	0 (0.00%)
Database Security	-	0 (0.00%)
Missing CWE	-	11 (2.17%)
Non-Vulnerable	-	454 (89.55%)
Total	-	507 (100.00%)

* CWEs added to the category in the updated database presented in this work.

7.3.1 Vulnerabilities and Approach

To demonstrate SCOLP, we make use of Linux Kernel, a *slice of the dataset* presented in Chapter 3. A summary of the number of code units (functions) per vulnerability category is shown in Table 7.4. For every function with a vulnerability previously reported at CVE Details [Özkan, 2023], we consider SMs and memory management-related attributes of both the vulnerable and the fixed versions.

After analyzing the SMs of the code units, we noticed that some of them have values that we consider too high. For instance, a function with more than 80 statements is hard to maintain, being more likely to have security issues [Walden et al., 2014]. After analyzing all functions, we established minimum and maximum values (*thresholds*) for some attributes. Functions with attribute values outside the threshold range are considered as not achieving a minimum quality criteria.

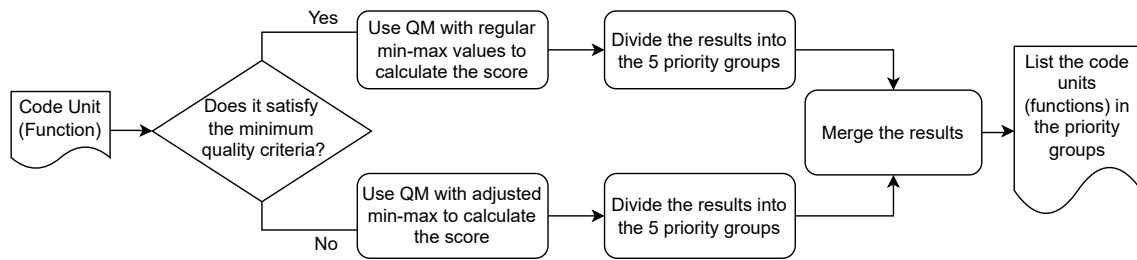


Figure 7.7: Workflow to validate if a code unit (function) satisfies the minimum quality criteria

Table 7.5: Configurations used in the experiments

Software Metric	Minimum Value	Maximum Value
CountStmt	2	80
CountLineCode	1	20
Cyclomatic	1	20
Knots	0	80

For the code units not achieving the minimum quality criteria, we shifted their priority to two groups to the left. In other words, the code units mapped into the *Lowest* priority group were moved to the *Medium* priority group. Also, the code units initially mapped to the *Medium* and *High* priority groups were moved to the *Critical* priority group. The reasoning is that code units not achieving the minimum quality criteria should not be in the low-priority groups. As for the code units that achieve the minimum quality criteria, the priority groups provided by the QMs are kept in the final mapping. The workflow followed to calculate the scores can be seen in Figure 7.7. The established thresholds can be seen in Table 7.5. It is important to highlight that functions with *CountStmt* smaller than 2 were discarded, as volume and complexity metrics cannot reveal information in this case, and these code units must be evaluated manually.

We defined the weights using the IDM, DWA, and WBR techniques explained in Section 7.1.1. The weights for the base QM (SMs as attributes) are on Table 7.6, while the weights for the specialized QMs (MM, IV, PER) are on Tables 7.7, 7.8, and 7.9, respectively.

7.3.2 Results and Discussion

This section presents the results, focusing on how code units (functions) were placed in the priority groups using the QMs with the configurations discussed previously. We aim to answer the following RQ:

- **RQ1:** Can we use SCOLP with QMs based on SMs and memory management-related attributes to characterize open-source code units (functions)?

Table 7.6: Weights used in the base SM QM

	QM node	IDM	DWA	WBR
V1	CountStmtDecl	0.33	0.35	0.33
	CountStmtEmpty	0.22	0.05	0.17
	CountStmtExe	0.44	0.65	0.50
V2	CountLineCodeDecl	0.28	0.30	0.30
	CountLineCodeExe	0.33	0.65	0.40
	CountLineComment	0.21	0.03	0.20
	CountLineInactive	0.19	0.02	0.10
C1	Cyclomatic	0.26	0.30	0.30
	CyclomaticStrict	0.34	0.35	0.40
	Essential	0.21	0.20	0.20
	Knots	0.19	0.15	0.10
V1_V2	V1	0.62	0.65	0.67
	V2	0.38	0.35	0.33
V1_V2_C1	V1_V2	0.39	0.40	0.33
	C1	0.61	0.60	0.67

Table 7.7: Weights used in the specialized MM QM

	QM node	IDM	DWA	WBR
MM	ALLOCATION_FUNCTIONS	0.52	0.55	0.67
	DEALLOCATION_FUNCTIONS	0.48	0.45	0.33
V1_V2_C1_MM	V1_V2_C1	0.41	0.45	0.33
	MM	0.59	0.55	0.67

The function distribution in the priority groups varies according to the QM configuration. Figure 7.8 shows the results for some QM instances. Two of them (*A* and *G*) have the majority of the functions in one priority group (Lowest or Medium). Others have a distribution in the lowest groups (*C* and *F*). On the other hand, some instances have one or more groups with few functions (*B*, *D*, *E*). Finally, some instances (*e.g.*, *H*) have a more even distribution among the priority groups.

To better understand the results, we further discuss four QM instances. The first one uses the IV model, and the function distribution can be seen in Figure 7.9. It uses IDM as the weight assignment technique and the simple arithmetic mean to aggregate the inputs. The vulnerable functions are mainly assigned to the most critical priority groups (Medium with 24 functions, High with 15, and Critical with 5). Note that the largest number of functions (both vulnerable and neutral)

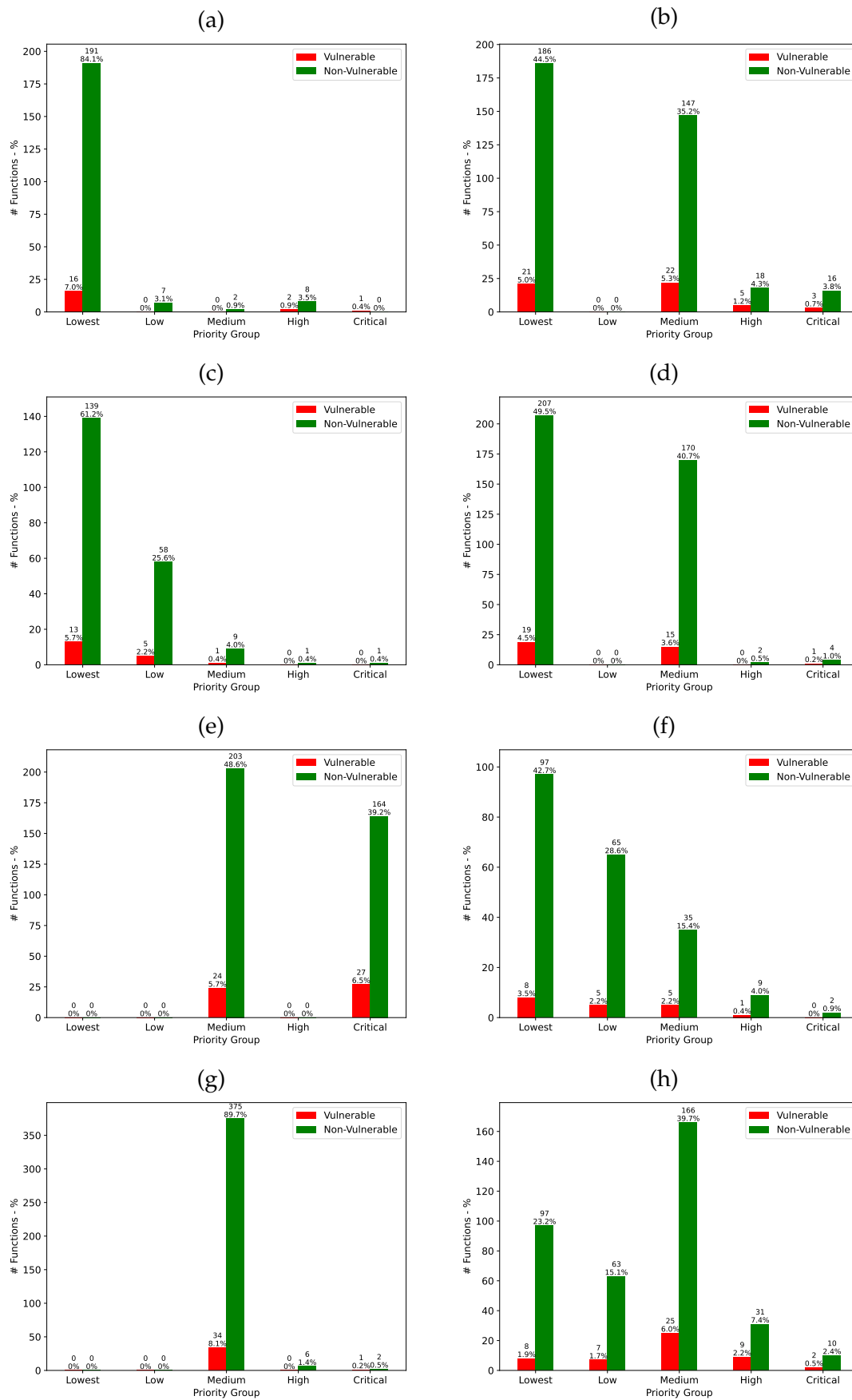


Figure 7.8: Examples of QM instances with different configuration

Table 7.8: Weights used in the specialized IV QM

	QM node	IDM	DWA	WBR
IV	CONVERT_UNSAFE	0.28	0.25	0.30
	STRING_UNSAFE	0.26	0.25	0.20
	SCANF_UNSAFE	0.30	0.25	0.40
	OTHER_UNSAFE	0.16	0.25	0.10
IV_MM	MEMORY_ADDRESS_OF	0.46	0.30	0.33
	IV	0.54	0.70	0.67
V1_V2_C1_IV_MM	V1_V2_C1	0.46	0.45	0.33
	IV_MM	0.54	0.55	0.67

Table 7.9: Weights used in the specialized PER QM

	QM node	IDM	DWA	WBR
V1_V2_C1_PER	V1_V2_C1	0.58	0.7	0.67
	MEMORY_ADDRESS_OF	0.42	0.30	0.33

in the Medium group is related to the shift that some functions had for not achieving the minimum quality criteria (191 out of the 418 functions with more than one statement).

The second configuration uses the PER model. Figure 7.10 shows the distribution of the functions in the priority groups. Unlike the previous configuration, it uses WBR for weight assignment but maintains the simple arithmetic mean to aggregate the score. Once again, most vulnerable functions are placed in the most critical priority groups (Medium with 24 functions, High with 17, and Critical with 5). As before, the Medium group has more occurrences due to the shift of functions that do not achieve the minimum quality criteria.

The third case is based on the IV QM, and Figure 7.11 shows the function distribution. Unlike the previous configurations, it considers only memory management-related vulnerabilities (functions with vulnerabilities in other categories are considered non-vulnerable). This configuration uses IDM and simple arithmetic mean to aggregate the scores. Most non-vulnerable functions are in the Medium priority group (39.95%), followed by the Lowest (22.25%) and Low (15.07%). On the other hand, most vulnerable functions (29 of the 35) are placed in the most critical groups (Medium, High, and Critical). This is expected, as they are the most critical from the security perspective compared to non-vulnerable functions.

The last QM configuration uses the PER QM, and the function distribution can be seen in Figure 7.12. As in the previous QM configuration, only functions with memory management-related vulnerabilities are considered vulnerable. Weights were defined using WBR, and scores were aggregated using the simple arithmetic mean. The groups with the largest number of non-vulnerable functions are the same as in the previous configuration (IV model): Medium, Lowest, and Low. As

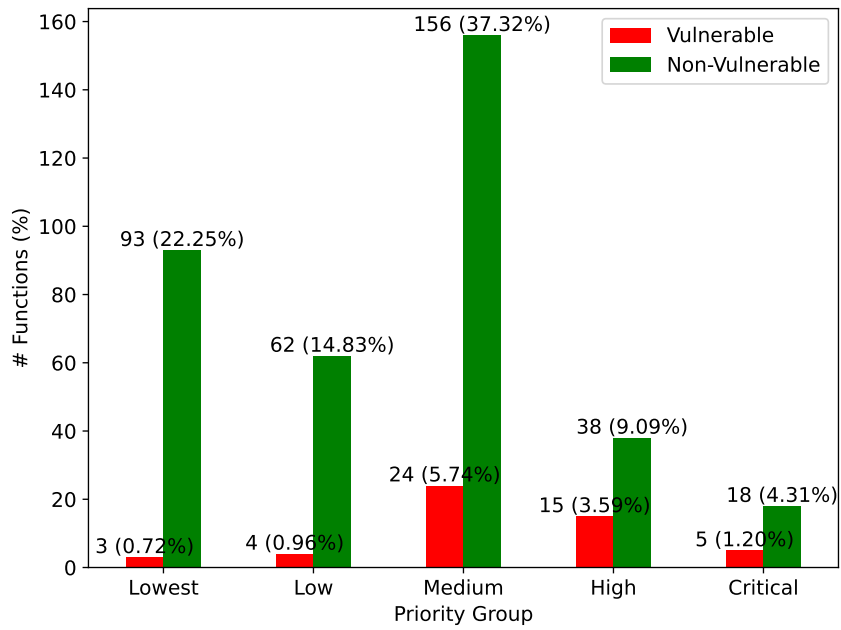


Figure 7.9: Function distribution - Quality Model IV

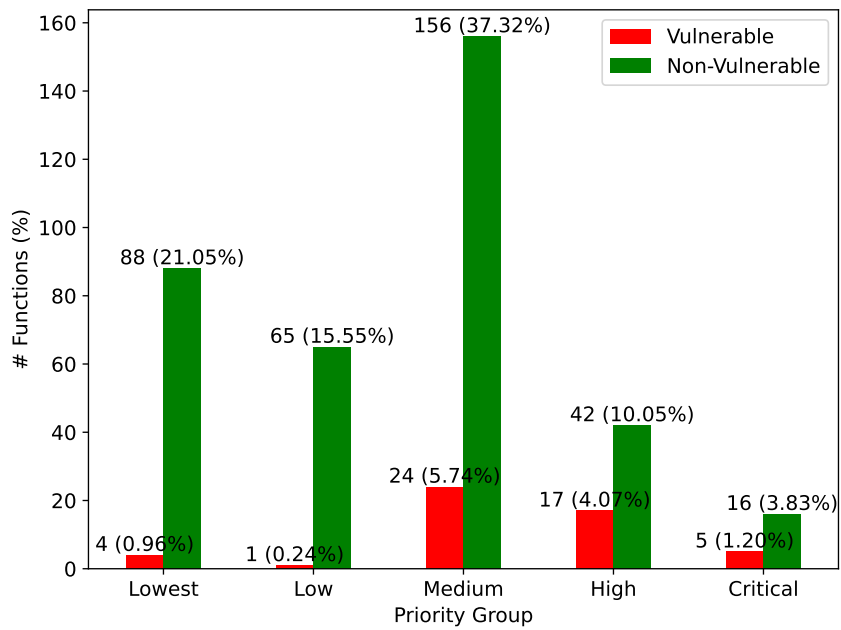


Figure 7.10: Function distribution - Quality Model PER

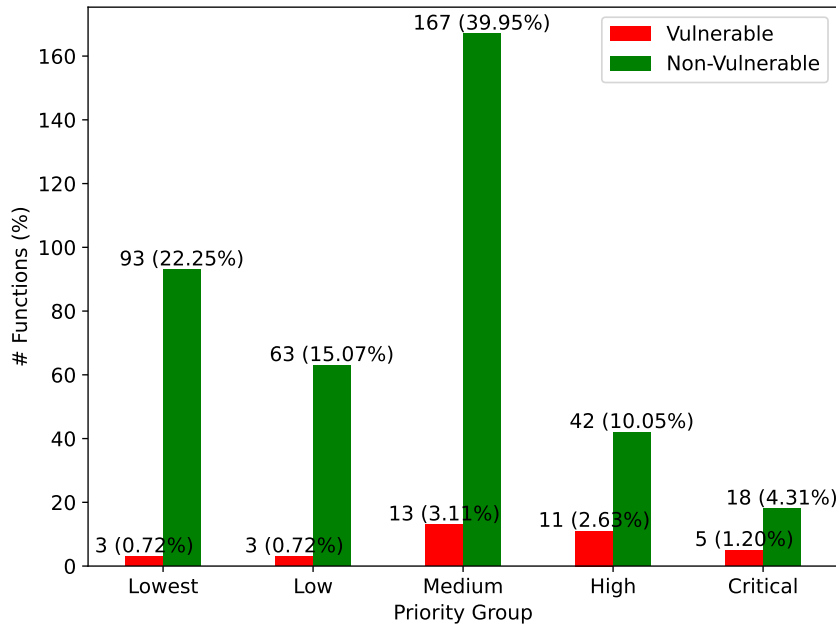


Figure 7.11: Function distribution - Quality Model IV. Only memory management-related functions are considered vulnerable

for the vulnerable functions, most of them (30 of the 35) are placed in the most critical groups (Medium, High, and Critical). Only 5 functions (about 16.66% of the vulnerable ones) are in the remaining groups (Lowest and Low).

The results above help us positively answering **RQ1**, as we can indeed use SCOLP with SMs and memory management-related attributes to characterize code units. Different from other techniques (*e.g.*, SATs and PTTs), whose goal is to detect vulnerabilities, SCOLP goal is to characterize code from a security perspective. Hence, a comparison with these techniques would not be fair. Although we have demonstrated SCOLP in a project developed in one programming language, it can be used for projects in other languages. It would require defining the attributes considering the language idiosyncrasies and following the process to calculate the scores accordingly. The following section presents a comparison of the SCOLP output with categorizations performed by experts.

7.4 Expert Ranking and Validation

This section describes the validation process with security experts. We aim to answer the following RQ:

- **RQ2:** Can SCOLP provide a categorization similar to security experts?

We start by introducing the experts that are part of the validation process; then, we explain the questionnaire answered by each expert; and, finally, we present the expert results and compare them with the categorization provided by SCOLP.

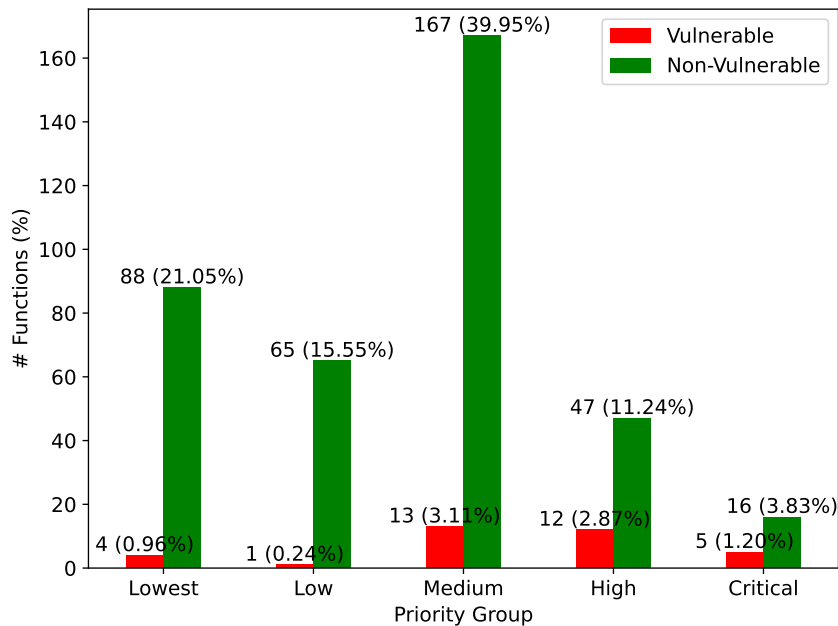


Figure 7.12: Function distribution - QM PER. Only memory management-related functions are considered vulnerable

7.4.1 Profile of the Experts

The experts that participated in this validation are researchers that work with software security: thirty-two holding a Ph.D. (one from the industry and the remaining from different universities), three Ph.D. Students, and one software engineer (with more than six years of experience). Although people from several countries and several affiliations were invited, not everyone accepted the invitation to answer the questionnaire. On the other hand, some participants forwarded the invitation to other colleagues and students. The countries and affiliations breakdown can be seen in Table 7.10. They are located in Europe (75.0%), South America (19.4%), and North America (5.6%). The questionnaire was anonymous, meaning identifying oneself was not mandatory. Hence, the total number of accepted invitations by country/affiliation is smaller (15) than the total number of responses (20) as we could not infer who provided 5 of the responses.

7.4.2 Questionnaire to Validate the Results

SCOLP can be used to characterize many code units fast as it is an automated process. Still, it was not viable to ask the experts to characterize all of them. Hence, we selected a small number (10) of functions to be analyzed, including five vulnerable and five neutral (non-vulnerable). We selected only functions with at least 50 lines of code to better understand if SCOLP can categorize similarly to experts in more complex scenarios. The functions selected can be seen in Table 7.11, as well as some metrics related to their size and complexity. The validation process is depicted in Figure 7.13.

Table 7.10: Invited experts by country and affiliation

Country	Affiliation	Invited	Accepted
Belgium	KU Leuven	1	0
Brazil	Federal University of Alagoas (UFAL)	1	0
	Federal University of Mato Grosso (UFMT)	1	1
	State University of Campinas (UNICAMP)	5	3
Canada	University of British Columbia	1	0
France	LAAS	1	0
Italy	University of Florence (UNIFI)	1	0
	University of Naples Federico II (UNINA)	4	1
Norway	Norwegian University of Science and Technology (NTNU)	4	1
Portugal	Codacy	1	0
	Critical Software	1	1
	Faculty of Sciences of the University of Lisbon (FCUL-UL)	2	0
	Polytechnic Institute of Guarda (IPG)	2	0
	Instituto Superior Técnico University of Lisbon (IST-UL)	1	1
	University of Coimbra (UC)	7	7
UK	City University of London	2	0
USA	University of Maryland	1	0
Total		36	15

The questionnaire was divided into two parts: (i) a categorization of the functions, and (ii) a pairwise comparison among the functions. In the first part, all experts received the ten functions and were asked to map them into one priority group (Critical, High, Medium, Low, or Lowest), defined in Section 7.1.3. A detailed explanation of the priority groups was provided to the experts. In the second part, the experts were asked to compare the functions pairwise with the priority they would put to improve/refactor them from a maintainability perspective. They received several pairs of two functions (*A* and *B*) from the first part and had to provide one of the classifications: *i*) *A* has more priority than *B*; *ii*) *A* and *B* have the same priority; and *iii*) *A* has less priority than *B*.

As the total number of pairwise comparisons of the ten functions would be too large for an expert to answer (45 comparisons), we reduced this number. Hence, we asked each expert to provide the 5 comparisons of 10 functions. As we wanted

Table 7.11: Description about the selected functions

	File - Function	Commit	# Lines	Cyclomatic
Vulnerable	(a) sound/core/timer.c - snd_timer_user_params	242658f	101	17
	(b) fs/fuse/dev.c - fuse_dev_splice_write	8fde12c	83	9
	(c) fs/splice.c - splice_pipe_to_pipe	8fde12c	111	13
	(d) fs/cifs/smb2pdu.c - SMB2_write	618d919	76	8
	(e) fs/ext4/xattr.c - ext4_xattr_set_entry	327eaf7	238	34
Non-Vulnerable	(f) fs/ext4/xattr.c - ext4_xattr_set_handle	5369a76	147	31
	(g) net/core/skbuff.c - skb_cow_data	8605330	91	12
	(h) fs/dcache.c - dentry_lru_isolate	946e51f	57	4
	(i) sound/usb/mixer.c - parse_audio_feature_unit	daac071	168	26
	(j) sound/usb/mixer.c - volume_control_quirks	daac071	111	32

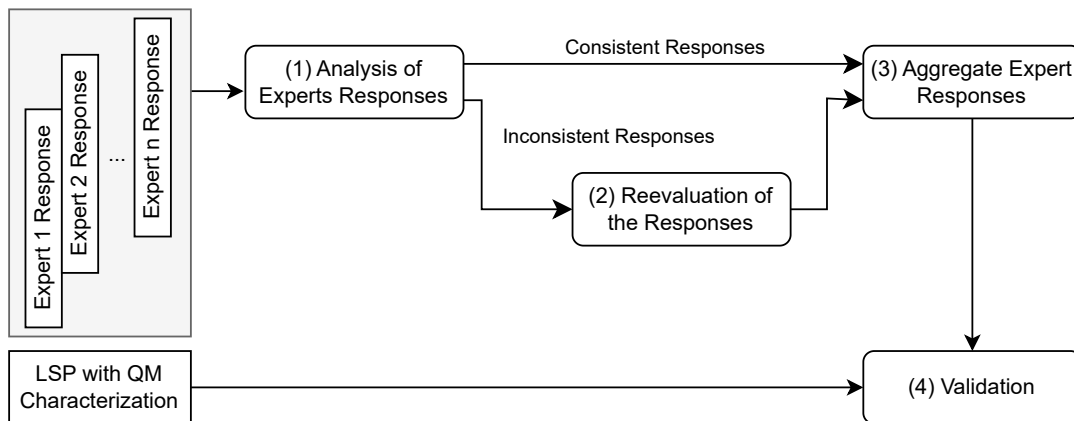


Figure 7.13: Validation process of the Expert Responses

all the comparisons, we would need 9 people to compare different pairs of functions, leading to the total number of expected comparisons (45). Furthermore, for every pair of functions to be compared at least twice, the minimum number of answered questionnaires is 18. We had 20 answers, and each questionnaire version was answered by at least two experts. Before sending the questionnaire to the experts, it was validated by another security expert (a Ph.D. student). We made some adjustments based on the obtained feedback, such as adjusting the questionnaire form to make the questions clearer to the respondents. These responses were not considered in the final results.

Experts were asked to classify the functions one at a time and to compare functions pairwise; thus, some inconsistencies happened. An inconsistency is found when the pairwise comparison contradicts the individual classification. For instance, an expert classifies a function *A* as *Critical* and a function *B* as *Medium* in the first part of the questionnaire. However, in the second part, the expert considers *A* less priority than *B*. We asked the experts with two or more inconsistencies to review their answers.

Table 7.12: Priority groups provided by QM instances and by the experts

Function	QM - IV	QM - PER	Lowest	Low	Medium	High	Highest	
Vulnerable	(a)	High	High	0	1	4	13	2
	(b)	High	High	0	4	2	9	5
	(c)	Medium	High	0	5	5	8	2
	(d)	Medium	Medium	1	7	9	1	2
	(e)	Highest	Highest	0	1	5	5	9
Non-Vulnerable	(f)	Highest	Highest	0	3	3	6	8
	(g)	High	High	1	3	7	5	4
	(h)	Medium	Medium	6	11	2	1	0
	(i)	Highest	Highest	0	1	8	6	5
	(j)	Highest	Highest	1	9	5	2	3

The **criteria used to consider the expert responses** were: *i)* no inconsistency is found in the questionnaire; *ii)* only one inconsistency is found in the questionnaire; and *iii)* two inconsistencies are found in the questionnaire, but at least one of the two functions is classified in the same criticality in the first part of the questionnaire (*e.g.*, functions *A* and *B* are classified as *High*, but in the second part, one of them has more priority than the other). In other words, if an expert has two inconsistencies and both have different criticality or if the expert has three inconsistencies or more, their response is considered inconsistent.

For the reevaluation process, we explained that at least one inconsistency was found (without disclosing details about where the inconsistency was), and they could change their answers. This process helps to reduce possible bias in the results. From the 20 answers, 6 experts had at least two inconsistencies. We could reach out to 4 of them, who revised their responses. The other 2 did not identify themselves, so they were not asked to revise their answers. Only one expert reviewed the individual classifications, but all 4 fixed the pairwise comparisons.

7.4.3 Experts vs SCOLP

The distribution in the priority groups made by the experts and those provided by the selected QMs can be seen in Table 7.12. For this analysis, we use the same QMs from Section 7.3.2 (the ones providing the more even distribution among the groups and the vulnerable files placed in the more critical ones). Note that the two QMs for IV (vulnerabilities related to Input Validation) and the two for PER (Permission vulnerabilities) provide the same results for the five functions under analysis. Thus, we present them once in the table to avoid repetitions.

As mentioned before, all the functions in the validation process have at least 50 physical lines of code. Because the QMs consider two groups of volume SMs (statements and lines of code), as shown in Figure 7.3, functions with more lines

have a higher score and a higher probability of being assigned to the more critical priority groups. As shown in Table 7.12, all the selected functions were assigned to the Medium, High, or Highest priority groups by the QMs.

Regarding the **vulnerable functions**, the selected QMs provided the same categories as most experts in most functions. The first two functions (*a* and *b*) listed in the table are assigned to the *High* priority group. This is the same as most experts. The same happens for the fourth (*d*) and fifth (*e*) functions, assigned to Medium and Highest priority groups by the QMs, respectively. Again, they are in the same groups assigned to them by the experts. The only difference is the third function (*c*), in which two QMs assigned to High (such as most experts), and the other two assigned to Medium (second most frequent priority group used by the experts).

All selected QMs provided the same classification for the **non-vulnerable functions**. The first non-vulnerable function (*f*) is assigned to the Highest priority group by all QMs and by the experts. The same does not happen to the other functions (*g*, *h*, *i*, and *j*), whose classification is impacted by the high number of lines of code. Nevertheless, they are assigned in most cases to a priority group adjacent to the one given by most experts (*e.g.*, the functions *g* and *h*).

These results help answer **RQ2**. Although the results provided by the QM instances are not the same as the ones provided by the security experts, they are similar. Hence, SCOLP can be used to provide a similar categorization as security experts without the need for their involvement.

7.5 Threats to Validity

This work addresses the issue of characterizing potentially problematic code units from a security perspective. Nevertheless, some threats should be discussed.

Construct Validity refers to the QMs used as LSP for the code unit characterization. A QM with SMs was defined and used as a base for the specialized QMs. While we considered attributes related to volume, complexity, and memory management, other attributes could also be considered. Although the results are consistent, different QMs (either different structure or attributes) could lead to better results. Additionally, the weights for the QMs were defined by the author of this thesis and were validated by his supervisor. Other options could be used to validate or explore different use scenarios.

External Validity refers to the ability to generalize the results. Only one project was used for the code unit characterization in our evaluation. Although it is a large project with an extended period since its development started, using a single project may affect the conclusions. In fact, QMs that are good enough for this project may work differently for others. Nevertheless, the approach can be customized considering the idiosyncrasies of each project and programming language. Additionally, a validation with security experts was performed. On one hand, this helps reducing this threat. On the other hand, the experts may

have different security knowledge, especially about C programming language, which can affect the results.

Internal Validity refers to the possibility of having unwanted or unanticipated relationships. From a security perspective, one or more attributes could lead to a more critical priority group. However, no other works in the literature could establish a relationship between attributes and their criticality or the presence of security issues. Hence, we can claim that this possibility is reduced.

7.6 Summary

In this chapter, we proposed SCOLP to characterize the security of code units (functions) using LSP. To do so, we defined QM instances that use SMs and memory management-related attributes as their attributes. Each QM calculates a score, which maps the code unit to a priority group. Consequently, the software development team gets categorized groups of functions considering their proneness to have security problems. The Linux Kernel project was used to evaluate SCOLP. We also performed a validation process with security experts. They classified ten functions using the same priority groups and compared them pairwise.

The results show that SCOLP can be used to categorize open-source code units (functions) using QM with different characteristics. Furthermore, SCOLP categorization is similar to the categorization provided by security experts. The QMs can be extended to address diverse code characteristics, enabling development teams to get tailored categorizations in an automated manner.

Although SCOLP can be easily integrated into CI servers to provide information while building the software, CI servers do not monitor the software at run-time. In the next chapter, we present a self-adaptive platform, named Trustworthiness Monitoring & Assessment (TMA), that can be used both at design-time (*e.g.*, to collect security evidence) or at run-time (to promote adaptation while the software system is being used). Although TMA was built before SCOLP, they share similarities (such as the QMs) and can be easily integrated.

Chapter 8

Framework to Promote Self-Adaptation

Trustworthiness can be defined as the worthiness of a service and its provider for being trusted [Medeiros et al., 2017], thus including a multitude of properties (e.g., reliability, availability, security, privacy, coherence, isolation, fairness, dependability, etc. [Mohammadi et al., 2013]). Continuously monitoring and assessing the trustworthiness of systems is not trivial due to many factors, such as the number of properties involved in trustworthiness. Also, trust is a subjective concept that is built based on guarantees, experiences, transparency, and accountability. However, current self-adaptive capabilities are rather limited and based only on *CPU usage* and *memory consumption* [Google, 2014a].

Self-Adaptive Systems (SASs) are capable of reflecting their own behavior and their environment and adjusting their behavior according to existing current needs [Krupitzer et al., 2015]. To promote self-adaptiveness, an adaptation control loop is normally added to the software system [de Lemos et al., 2013]. The most used adaptation control loop was introduced by IBM and is named *MAPE-K* [IBM, 2006]. Its name is an acronym for five components: Monitor, Analyze, Plan, Execute, and Knowledge. The *Monitor* is responsible for collecting details about the managed resource, while *Analyze* reasons over the data collected in the previous phase. Decisions are made in the *Plan* to achieve the goal and objectives. *Execute* is responsible for interacting with the *managed element* to ensure that the adaptations are made. Finally, *Knowledge* is a repository that supports other phases. TMA implements all MAPE-K components, allowing other systems to have self-adaptive properties when using TMA.

This chapter presents the **Trustworthiness Monitoring & Assessment (TMA)** platform, which brings self-adaptation abilities to software applications and contributes to the **assessment and improvement of the trustworthiness**, considering the relevant properties and a trustworthiness life cycle inspired in the MAPE-K cycle [IBM, 2006]. The TMA platform supports these activities by providing a solution for **applications that require self-adaptation for maintaining/achieving trustworthiness without the need for creating a managing element from scratch**. Compared to existing solutions (e.g., HPA [Google, 2014a]), our proposal

adds flexibility and allows the user to prepare their systems to adapt according to a wide range of properties. The interface between the application and TMA is supported by *probes* to monitor the status of the application and by *actuators* to perform the adaptations [IBM, 2006]. Quality Models (QMs) aggregate data collected from the cloud application and generate trustworthiness scores. A basic trustworthiness QM is explained in [Medeiros et al., 2017]¹.

Each TMA component is designed as a microservice [Lewis and Fowler, 2014] that can be easily deployed in a container-based system (*e.g.*, Kubernetes, Docker Swarm). A usage scenario shows the applicability and flexibility of the platform. In practice, we monitor resource consumption (CPU usage and memory consumption) and performance (response time and throughput) of a cloud application. TMA calculates its trustworthiness level, and adaptations are made when needed. The scale up/down actions of the application server are dispatched according to the workload. Results allow observing how the scores can guide the system adaptation. Further information about TMA can be found online: <http://tma.dei.uc.pt/>

It is important to emphasize that TMA was used in the context of the ATMOSPHERE project² by the Dell company. A QM for security was designed for their context, focusing on sub-properties based on the security triad (confidentiality, integrity, and availability). A probe developed by Dell collects configuration properties and gathers security policies. If the score calculated using the QM is below a given threshold, the security team is alerted to act.

The remainder of this chapter is structured as follows. Section 8.1 presents concepts on SASs. Architecture and implementation details are presented in Section 8.2. The usage scenario is presented in Section 8.3 and the results and discussion in Section 8.4. Finally, Section 8.5 summarizes this chapter.

8.1 Concepts on Self-Adaptive Systems

There are two components in a Self-Adaptive System (SAS): *i*) the target of the adaptation called *manage element*, and *ii*) the *managing element* responsible for promoting the adaptation. The adaptation control loop is implemented in the *managing element*. All the interaction between the managing element and the managed element happens through the manageability endpoints. *Sensors* or *Probes* are used to send all the data from the managed element to the managing element. *Effectors* or *Actuators* perform the adaptations on the managed element.

There are different approaches to promote self-adaptation, such as the architecture-based solution used by Rainbow [Garlan et al., 2004]. It adds the

¹ Similarly to SCOLP (presented in the previous chapter), TMA uses QMs to support the decision-making process. However, while the prior uses them to categorize code units at design time, the latter can also support the adaptation of running applications (in addition to design time aspects). Although SCOLP and TMA are not integrated, such integration would be straightforward due to the TMA architecture.

² <https://www.atmosphere-eubrazil.eu>

adaptation control loop in the architectural layer, while probes and effectors stay in the system layer along with the managed element. The architecture-based self-adaptation allows a global perspective of the system, and system-level properties and integrity constraints are exposed.

Several self-adaptation features are found in a cloud environment. For instance, the infrastructure provider Amazon Web Services has the AWS Auto Scaling service [Amazon, 2009], which provides a rule-based autoscaling service for the resources deployed on the Amazon Cloud. For container-based systems (*e.g.*, Kubernetes [Google, 2014b], Docker Swarm [Docker, 2017], Apache Mesos [Apache, 2009]), orchestrator frameworks are usually responsible for autoscaling features. For instance, Horizontal Pod Autoscaler (HPA) [Google, 2014a] for systems deployed using Kubernetes. Both Amazon AWS and Kubernetes HPA usually use metrics of memory consumption and CPU usage to support the scaling.

Our solution differentiates from existing ones in terms of flexibility (of monitoring and adaptation and on supporting diverse systems). Also, we use Quality Models (QMs), such as the ones presented in the previous chapter, to aggregate diverse metrics according to user-defined preferences, and that simplifies the task of decision-making based on several properties. In practice, it is useful in scenarios that require: *i)* more diverse or more complex adaptations; *ii)* the analysis of diverse trustworthiness properties and scores (*e.g.*, security, privacy, dependability, and coherence); and *iii)* the computation of scores that require inputs from multiple levels of the managed element (*e.g.*, at the client level, service level).

8.2 TMA Platform Architecture

The TMA platform supports the trustworthiness assessment that involves different attributes, properties, and characteristics, which vary depending on the objectives of the system that is being assessed [Medeiros et al., 2017]. Our trustworthiness framework for applications is composed of:

- A definition of the relevant properties and metrics used to characterize trustworthiness
- A trustworthiness lifecycle (Figure 8.1), inspired in the MAPE-K cycle and that covers two main phases: design-time, when the application is being developed, and run-time, when the application is being used
- A monitoring platform that receives measurements and events from the managed application
- Measurement instruments that allow gathering the information to be used (measurements and events)
- Quality Models (QMs) that define how the measurements from the attributes will be used to compute the scores (more details on how to define a QM can be seen in the previous chapter)

- Actuators that implement the adaptation logic are included in the managed system and allow adaptations that aim at improving the system trustworthiness

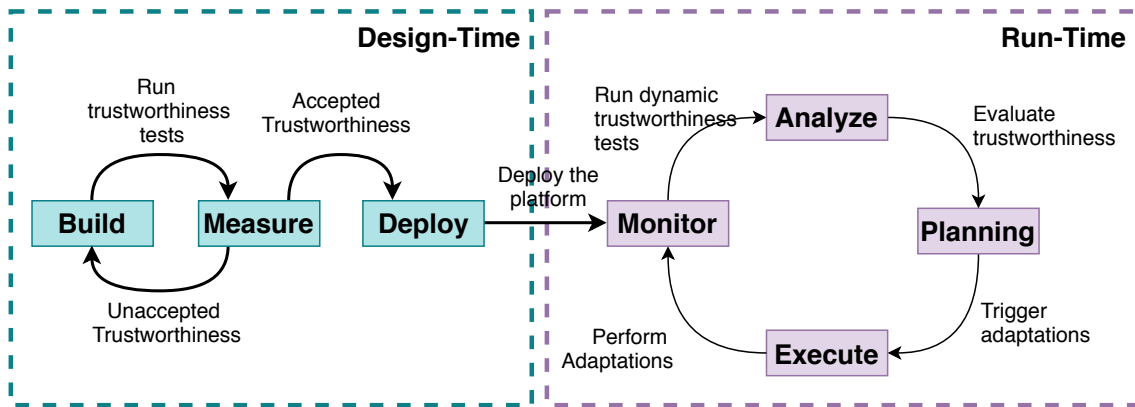


Figure 8.1: Lifecycle of Trustworthiness Assessment

Figure 8.2 presents a high-level architecture of the TMA platform, which follows a microservice architecture [Pahl and Lee, 2015], where each component represents one MAPE-K function (Monitor, Analyze, Plan, Execute, and Knowledge) [IBM, 2006]. Each one of the components is deployed into a container inside a Kubernetes pod. Kubernetes is an open-source system that allows automatizing the deployment and management of container-based applications [Google, 2014b]. Kubernetes starts the containers and deploys them according to the specification of the pod (wrapper defined by Kubernetes and its base unit of management).

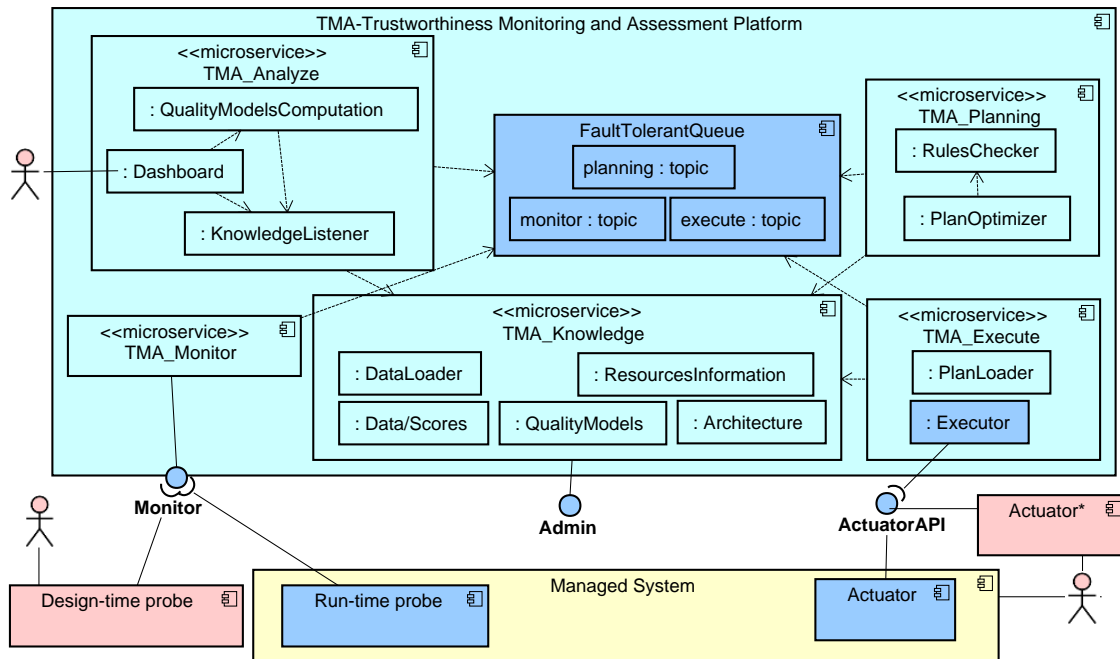


Figure 8.2: Architecture and Interfaces of TMA

The implementation details of each component are in the following sections. To ensure that the communication among the components is handled in a reli-

able way, a fault-tolerant mechanism is used. The `FaultTolerantQueue` is implemented using Apache Kafka, which allows creating topics that the components can subscribe to consume the messages [Apache, 2011]. Details on how to create a probe and an actuator are described in Sections 8.2.1 and 8.2.4, respectively. A Docker image and a YAML (YAML Ain't Markup Language) file were created for each component of the TMA to support the deployment into a Kubernetes cluster. The YAML file contains the specification of a Kubernetes object, and Kubernetes uses it to decide what to deploy (e.g., the image container that will be used, and the ports that will be open to be invoked). We automate the platform deployment through bash scripts and recipes (<https://github.com/eubr-atmosphere/tma-yaml>).

8.2.1 Monitor Component

The **Monitor** component provides a RESTful API interface for the *probes* to post JSON messages (observations) of the managed element (i.e., the application) to TMA. This component is responsible for validating all data collected by probes according to a JSON schema. The service is deployed using the web microframework Flask [Ronacher, 2010]. TMA is prepared to receive the results of both design-time assessment tasks (in which performance and scalability requirements are not so stringent) and run-time measurements (when the behavior changes according to the system execution).

This component validates all data collected by probes according to a JSON schema. SSL/TLS encryption is used to secure communication between the probes and the Monitor component. Hence, all probes must have the Monitor digital certificate acquired during the initial registration of the probe in the platform. Later, when a probe sends data to the Monitor, its certificate is used to authenticate.

The JSON schema is used to validate data received from the probes. The content of the message contains also its data type, which can be: *i) measurement*: any numerical value that can be assessed (e.g., execution time, memory allocated, CPU in use, response time); *ii) event*: any occurrence that needs to be stored (e.g., the scale up process has started). Additionally, both the value and the time of the measurement are sent. If the data received are valid, the Monitor sends them to the `FaultTolerantQueue` through an Apache Kafka topic named `monitor`. Otherwise, the data are discarded, and an error message is returned to the probe.

The `QueueListener` subcomponent is implemented through the `DataLoader` component. It pulls data from the `monitor` topic and executes the data normalization process, so the data are correctly inserted in the Knowledge database.

A few probes were developed to demonstrate and validate the TMA platform, and libraries to develop new probes were created for Java, Python, and C#, as well as a Docker image. Using the supporting tools described above, we make available 12 concrete probes for different measurements, and information about them can be found online (<http://tma.dei.uc.pt/probes>). The ones listed below are used in the usage scenario (see Section 8.3):

- `probe-k8s-metrics-server` – gathers information about pods and nodes in a Kubernetes cluster. It collects measurements of *CPU usage* and *memory consumption* using the monitoring open-source component *metrics-server* (<https://github.com/kubernetes-sigs/metrics-server>), which replaces the deprecated Heapster [Google, 2014b].
- `probe-client-java` – collects performance metrics of a client that performs REST requests to a server. It is able to monitor *response time*, *throughput*, and *rate of served requests under a predefined threshold*.

8.2.2 Analyze Component

The **Analyze** component is responsible for reasoning over the data gathered by the Monitor component and aggregating the measurements into a trustworthiness score. The data are read from the Knowledge component, and the scores are calculated and stored. Several aspects may be considered to support the decision-making (adaptation) process. Hence, a combination of different sources of information is necessary.

Quality Models (QMs) allow aggregating measurements and computing a score. The focus of the Analyze component is on trustworthiness properties, as this is the main goal of the platform. Thus, QMs for different properties (*e.g.*, security, privacy, coherence, isolation, stability, fairness, transparency, dependability, etc.) can be used to compute trustworthiness scores [Mohammadi et al., 2013]. A Dashboard that allows users to analyze information at runtime and adjust the parameters of the QMs and thresholds (objectives) is also available.

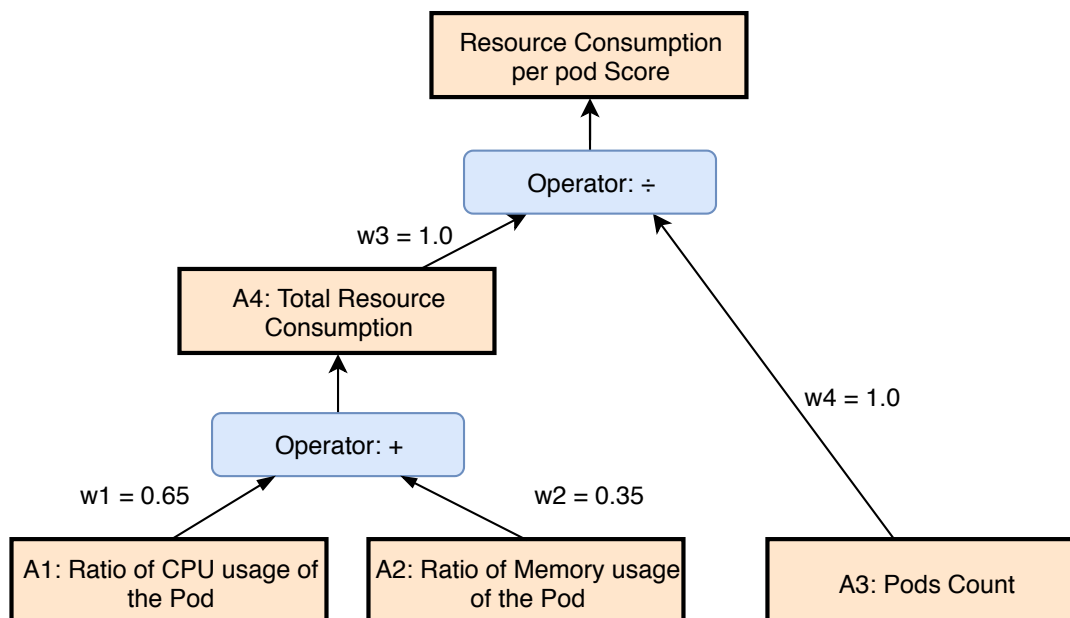


Figure 8.3: Resource Consumption QM used in this practical demo

As an example, the score of *resource consumption per pod* QM is defined in Figure 8.3. The leaf nodes (A1 and A2) show ratios of both CPU usage and memory

consumption of a pod in comparison to the working node where the pod is deployed. The ratios are calculated by dividing the value obtained from the probe (either CPU usage or memory consumption of the pod at a particular time) by the total available in the working node (total CPU and memory available, respectively). They are aggregated into a metric of Total Resource Consumption (A4). In this scenario, CPU Usage (A1) has more weight ($w_1 = 0.65$) than the Memory Consumption (A2) ($w_2 = 0.35$), as CPU usage is more sensitive when the application server receives a higher workload. Finally, the Total Resource Consumption Score is divided by the leaf node Number of Pods (A3) to compute the final score. All the attribute weights are defined using the DWA method. As explained in Section 7.1.1, the attributes can be of two types: benefit or cost attribute. The attributes are represented in Figure 8.3 with different colors: benefit attributes in green and cost attributes in orange.

The partial values of the leaf nodes A1 and A2 can be summarized by the following formulas:

$$A1 = \text{cpu_pod} / \text{cpu_node}$$

$$A2 = \text{memory_pod} / \text{memory_node}$$

where both *cpu_node* and *memory_node* are constants known in advance. The score *resource consumption per pod* can be calculated by the following formula:

$$\text{score} = (0.65 * A1 + 0.35 * A2) / \text{pod_count}$$

The Analyze component needs to know the following configurations in advance: *i) periodicity of calculation*, and *ii) observation window*. The observation window defines the amount of data in a period used to calculate the score. The periodicity of calculation defines how often the score will be calculated. For instance, a score is calculated every minute using the data from the last ten minutes. In this case, the observation window is ten minutes, while the periodicity of calculation is once per minute.

8.2.3 Planning Component

The **Planning** component is responsible for checking the scores and coming up with a plan in case an adaptation is needed. An adaptation plan is a set of actions to achieve the required goals or to recover the desired trustworthiness levels. There are different adaptation decision approaches, *e.g.*, models, rules/policies, goals, or utility [Krupitzer et al., 2015]. A rule-based approach [Macías-Escrivá et al., 2013] is used by the TMA.

TMA uses the business rules management system *Drools* [RedHat, 2011], which is a Java-based tool. It provides an easy-to-read and easy-to-understand language to specify rules, with conditions and actions in case the conditions are met. The

decision to adapt is made based on the score calculated by the *Analyze* component. If the score is either above or below a threshold, an adaptation is dispatched. In case the user needs to extend it, a Java class can be created and invoked by the specified rules.

An example of the Drools rule used by TMA is shown on Listing 8.1. The *performance* score (calculated using the QM detailed in Section 8.2.2) is used to specify the condition through the `when` directive. When the score value exceeds the threshold of 0.08, and the number of pods is smaller than 2, an adaptation is dispatched to increase the number of pods to 2. The adaptation plan is specified through the `then` directive. Details about the execution of the plan are presented in Section 8.2.4.

```

1 | rule "Score validation - Wildfly Scale up"
2 | when
3 | $score: TrustworthinessScore ( performanceScore.score > 0.08, podCount <
4 | 2 )
5 | then
6 | Action action = new Action(1, "scale", 9, 5);
7 | action.addConfiguration(
8 | new Configuration(
9 | 2, "metadata.name", "wildfly"));
10 | action.addConfiguration(
11 | new Configuration(
12 | 3, "spec.replicas", "2"));
13 | AdaptationManager.performAdaptation( action, AdaptationManager.
    obtainMetricData($score) );
    end

```

Listing 8.1: Scale up Drools rule example

A similar rule is needed to scale down the number of pods. In that case, a different, lower-bound threshold should be used.

8.2.4 Execute Component

The **Execute** component is responsible for communicating with the Actuators to perform the adaptations. This is done through RESTful operations. All communication is secure, and all the messages are encrypted using the keys of both the Execute component and the actuator to be used.

The Execute uses the public key from the corresponding actuator to encrypt the message (which is obtained when the actuator is registered in the platform). The public key is stored by the Knowledge component. When the actuator receives an encrypted message, it uses its private key to decrypt it. Then, the actuator performs the actuation and responds to the Execute. This message is encrypted using the public key of the Execute, and it is signed using the private key of the Actuator. When the Execute receives the response, it verifies the signature using the public key of the actuator and decrypts the message using its private key.

Currently, two libraries are provided to ease the communication with actuators developed in Java and Python. Three actuators are available: *i*) `kubernetes-actuator`: used to scale up and scale down Kubernetes pods; *ii*) `email-actuator`: used to send e-mail notifications when the scores are not in the

expected thresholds; and *iii*) *api-actuator*: used to interact with a third-party API.

The actuator *kubernetes-actuator* is used in the practical demonstration presented in Section 8.3, and it was developed using the Actuator Java Library. It receives a JSON object, which needs information to perform the action, including the parameter values. The values presented on Listing 8.1 of Section 8.2.3 are mapped into this attribute).

8.2.5 Knowledge

The **Knowledge** component is responsible for storing all the data, such as measurements and events, QM definitions, trustworthiness scores, information about the application architecture, resources and assets available, and adaptation plans. Its implementation contains a MySQL DBMS (*knowledge* database) and a block-storage solution Ceph.

Data collected by the probes are inserted in the corresponding table. The TMA database follows a star schema as the one shown on Figure 8.4. The fact table *Data* contains the basic numerical facts provided by the probes, and the dimensions include all the different perspectives needed to characterize them. The remaining information about the architecture is represented in the *Resource* and *Probe* tables. The *Description* table contains the specification about either the measurements or events provided by each probe. Finally, the *Time* dimension is represented in the conceptual diagram, but the information about the time is stored directly in the *Data* table (for performance reasons).

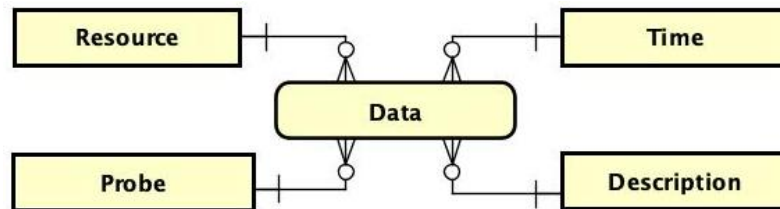


Figure 8.4: Data model used by TMA for the Monitor Component

The model used by the *Execute* component reflects the data to dispatch the adaptations (see Figure 8.5). Information about the actuator, as well as the actions that they can perform, is stored in the tables represented in the adaptation model. The *Actuator* table stores all the active actuators, and the possible actions are stored on the *Action* table. The *Configuration* table contains the specification of additional parameters needed by the actions. The action is related to a resource, which is the target of the adaptation.

An application called *Admin* was developed to ease the configuration of the platform via the knowledge database. It allows registering probes and actuators to the database, besides information about the architecture of the managed element. It is split between two different components: the REST API and the GUI. Both components are deployed as a Kubernetes pod.

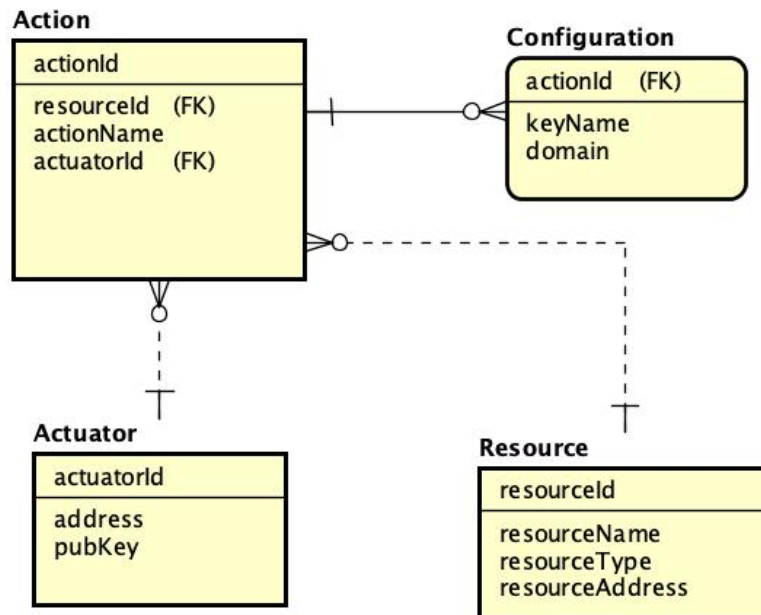


Figure 8.5: Data model used by TMA for the Actuators

The REST API is implemented in Java, using the Spring framework to expose the REST services. The API takes care of every request and proceeds to update the Knowledge database. The GUI provides a user-friendly interface that allows the TMA administrator to invoke the REST API service. Features such as adding a probe, adding an actuator and configuring their actions, and adding the resources of the managed system are available in this application.

8.3 Usage Scenario: Scaling Containers

The usage scenario presented in this section shows one application that consists of a set of web services from TPC Benchmarks [TPC, 2014]. To store the data, both the web application server Wildfly [RedHat, 2014] and MySQL database are used. We focus on performance-related experiments as these are the ones that have alternatives even if limited (*e.g.*, HPA), and they are easy to understand. However, TMA can be used with other metrics such as dependability, privacy, and security.

The experiment consists of varying the request workload to the application and observing the adaptations and the scores. Four different scenarios are used:

- A: No adaptation approach
- B: Using Kubernetes HPA [Google, 2014a]
- C: Using our proposal with Resource Consumption QM (Figure 8.3)
- D: Using our proposal with Performance QM (Figure 8.7)

The application is deployed on Kubernetes. It is configured using the controller StatefulSet from Kubernetes, which guarantees the order in which the pods are created and scaled. As it is using the StatefulSet controller, every time the number of pods changes through a scale action, a pod is either created or deleted when the remaining ones are running properly to avoid disruptions in the service. This solution allows scaling the number of pods, as the requests are sent to the same endpoint. All the load is balanced among the pods by Kubernetes.

An example of adaptation is shown in Figure 8.6. The hexagon represents the working node where the pods are deployed. A working node is a machine (either physical or virtual machine) that contains the resources to run pods. Although all the pods are represented in one working node, this is not known in advance since Kubernetes is responsible for assigning the pods to the working nodes. The dashed ellipse represents the Wildfly service. The blue circles represent the pods, and the green cubes show the containerized applications. Initially, only one pod is created. When the workload increases, more pods are created to balance the load. As the MySQL database does not scale during the experiment, it is not represented in the diagram. It is also deployed in a StatefulSet with one pod.

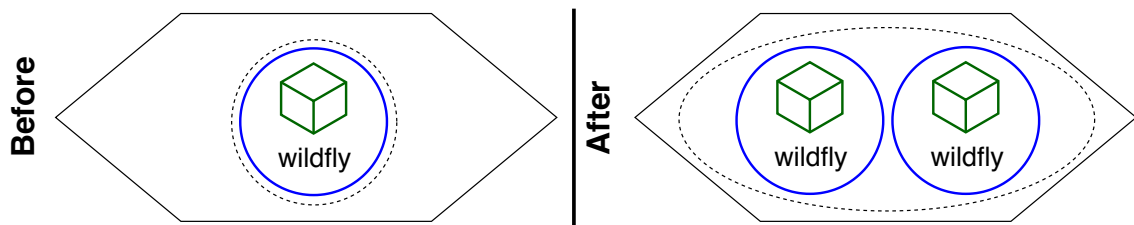


Figure 8.6: Example of scaling up the number of pods (before and after the adaptation)

All the experiments were performed in a machine which has the following configuration:

- **CPU:** Intel(R) Xeon(R) Gold 5118 CPU 2.30GHz x 24
- **Memory:** 96GB DDR4 RAM
- **Disk:** Dell SSD PERC H330 Adp 1TB

This server, used to create Virtual Machines (VMs), uses the tool Infrastructure Manager (IM) [Caballer et al., 2015]. IM automates the deployment and configuration of VMs. For this practical demonstration, four VMs were created for the Kubernetes cluster:

- **Control Plane:** instance that contains the control plane of the Kubernetes cluster. It allows creating the Kubernetes objects (*e.g.*, nodes, pods, volume storage)
- **Working Node (2 instances):** instances where the pods with Docker containers are deployed and run. After joining the cluster through the control plane, pods can be deployed in the nodes by the control plane

- **Storage Node:** an instance where the block storage is configured (Ceph [Weil et al., 2006]). As data stored in the pods are ephemeral, there is a need for a persistent solution for the data

The configuration of all instances has 4 CPU cores, 16GB of Memory, and 200GB of disk storage.

When both the managed element and the TMA are properly set up, the experiment is started. The scenario consists of varying the demand of requests to the application (requests per second - rps) for 30 minutes. Each slot run during three minutes, and the configuration of each one can be seen in the Table 8.1.

Table 8.1: Slots of demand in the experiment

Slot	Demand (rps)
I	150
II	300
III	650
IV	1000
V	2500
VI	2000
VII	650
VIII	1000
IX	650
X	300

During the phases of the experiment, two scores were calculated: *i*) the *Resource Consumption per Pod Score* (according to the QM defined in Figure 8.3), and *ii*) the *Performance Score* (according to the QM defined in Figure 8.7). They are used to decide when scale up or scale down actions are needed.

8.3.1 Resource Consumption per Pod Quality Model

This is the same QM that was defined in Section 8.2.2. The probe `probe-k8s-metrics-server` pushes the data to RESTful interface of the Monitor component every 5 seconds. It is important to notice that the CPU usage and memory consumption weights ($w1 = 0.65$ and $w2 = 0.35$) were empirically defined by using the DWA method. It was noticed that CPU usage varies more than memory consumption when the workload increases, and this is the reason the CPU usage weight is larger than memory consumption.

The score is calculated every five seconds (*periodicity of calculation*), providing an up-to-date status of the managed element. The tool used to collect CPU usage and memory consumption does not provide accurate information more often. As all the attributes from the Resource Consumption per Pod Score are cost attributes (represented with orange color in the figure), the final score can also be interpreted as a cost score. In other words, the higher the score, the worse from a resource consumption perspective.

8.3.2 Performance Quality Model

The measurements used by the Performance QM are obtained through the probe-client-java probe (described in Section 8.2.1). Different from the probe-k8s-metrics-server probe, which is deployed with the application, this probe is deployed on the client side, and it collects measurements from the user perspective. The leaf attributes contain either values obtained from the probe (e.g., *throughput* - A1, *response time* - A2, or *rate request under contracted* - A3), or obtained from the calculation of values by the probes (e.g., *rate served requests* - A4, which is the division of throughput by demand).

Among the leaf attributes, some of them are benefit attributes (*throughput* - A1 and *rate served requests* - A4), and others are cost attributes (*response time* - A2 and *rate requests under contracted* - A3). They are represented in Figure 8.7 with different colors: benefit attributes in green and cost attributes in orange. Hence, an adjustment needs to be made to combine them.

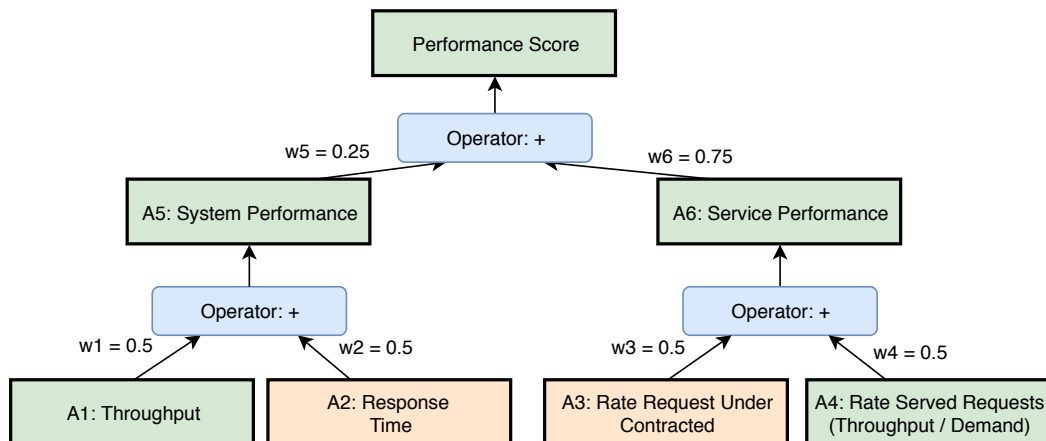


Figure 8.7: Performance QM used for this practical demonstration

From a semantics perspective, it makes sense to have a performance score as a benefit attribute. Therefore, the adjustment needs to be done in the cost attribute to be interpreted as a benefit attribute. As all the attributes are normalized and are in a 0-to-1 range, the transformation is done through the following formula: $benefit_attribute = 1 - cost_attribute$. After the transformations, the attributes can be combined, resulting in another benefit attribute.

The first partial score is *System Performance* (A5). It is composed by the *throughput* (A1), and *response time* (A2), and they have equal weight ($w1 = w2 = 0.5$). Similarly, the second partial score is the *Service Performance* (A6). Its attributes (*rate request under contracted* - A3, and *rate served requests* - A4) also have the same weight ($w3 = w4 = 0.5$) and are aggregated through the sum operator.

To come up with the final performance score, the neutrality operator is also used. However, partial attributes have different weights. As the *Service Performance* partial score is more important, it received the weight $w6 = 0.75$. This way, the *System Performance* partial score has a weight of $w5 = 0.25$. This score is calculated every second. As the probe sends data to the Monitor component only when it is running, and due to the adjustment made on the cost attributes, it was necessary

to adjust the score when no data is being informed to the platform. Hence, this value is set to zero when no data are present from the client side.

8.4 Results and Discussion

This section presents the results of each run per configuration of the demonstration application based on TPC Benchmarks. Figure 8.8 (a) shows the run chart of the scores when no adaptation mechanism is enabled. The blue line presents the *resource consumption per pod* score, which varies based on both CPU usage and memory. Looking closely at the data, we can observe that the score is linearly related to CPU usage, which is influenced by the demand (rps). The *performance score* is presented in red, as well as their partial sub-scores (dashed lines). Between instant 105 and 160, the performance score drops, as one of the replicas cannot handle a load of 2,500 rps.

Differently from the previous scenario (Figure 8.8 (a)), Kubernetes HPA [Google, 2014a] is used as an adaptation approach, and one chart is shown on Figure 8.8 (b). As it can be seen on the *performance score*, it does not vary so much, although there are some peaks. Throughout this experiment, a new pod is created. However, even when the demand decreases, and when no demand is present, the number of pods remains equal to two. Regarding the *resource consumption per pod score* line, there is a peak around instant 61. This is related to the scaling promoted by HPA, which happened some instants earlier. When a new pod is created, both CPU usage and memory are high. Consequently, the score increases. When the pod set up stabilizes, the values also go back to normal.

The remaining run charts in Figure 8.8 (c and d) show the results when the adaptation is dispatched by the TMA platform. Every time a score calculated through a QM is either above an upper threshold or below a lower threshold, an adaptation plan is created. The adaptation plan is executed by the Execute component, which invokes the `kubernetes-actuator`. It is first invoked when the score exceeds the upper threshold.

Figure 8.8 (c) shows the scores during one execution. When the adaptations are performed using the resource consumption QM, four adaptations are performed (two scale up, two scale down). The two peaks on the *resource consumption per pod score* chart represent the creation of the new pods. Differently from the adaptation with HPA, the scale down happens. Thus, the scores fall when no demand is being performed, and the resources are not allocated when they are idle. Also, the *performance score* does not vary so much, but there are some peaks during the higher load of the experiment.

The adaptations triggered by the performance QM have similar behavior as the resource consumption per pod QM. Figure 8.8 (d) shows the scores during an experiment execution using the performance QM, and the peaks on the *resource consumption per pod score* chart represent the creation of the new pods. As the *performance score* is being used to decide about the adaptation actions, it varies more during the experiment.

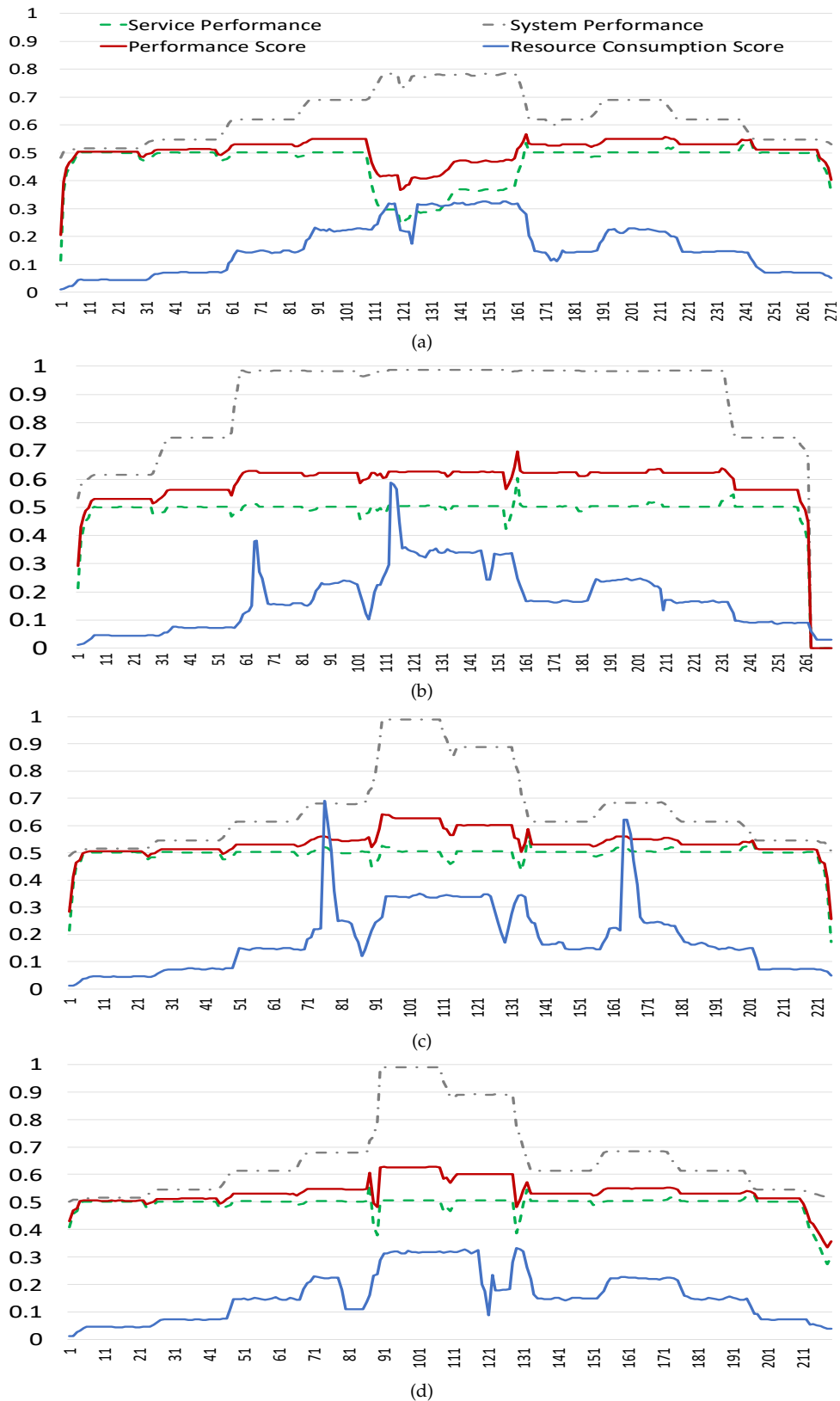


Figure 8.8: Charts of the Resource Consumption per pod and Performance Scores during one experiment of the TPC Benchmark usage scenario: (a) without adaptations; (b) with adaptations dispatched by HPA; (c) adaptations using the Resource Consumption per pod QM; (d) adaptations using the Performance QM

Table 8.2 shows the average response time (top of the cell) and the average throughput (bottom of the cell) per slot configuration. The stars (*) represent the slots when an adaptation happens (either scale up or scale down). It can be noticed that only one adaptation happens in the HPA configuration, while the adaptations using the platform scale up and scale down during the experiment.

The response time values show that the mean is higher when no adaptation is performed. In configurations B, C, and D, the response time is lower because a new replica is created, which means that the requests are distributed by the two replicas. This decrease can be observed with more impact in slots V and VI when the response time decreases by about 2.5 milliseconds.

The throughput values show that the mean is higher in configurations that automatically create a new replica of the service. With two replicas, the service can handle loads of slots V and VI (2,500 and 2,000rps, respectively), which increases the mean in configurations B, C, and D in comparison with configuration A.

Table 8.2: Average response time and throughput of each configuration (A - No Adaptation, B - HPA, C - Resource Consumption, D - Performance)

Slot	Demand	Response time (ms) / Throughput (rps)			
		A	B	C	D
I	150	4.64	4.77	4.58	4.59
		149.94	149.97	149.97	149.97
II	300	4.37	4.42	4.54	4.50
		299.82	299.89	299.62	299.53
III	650	4.68	5.22	4.85	4.85
		649.62	649.92 *	649.93	649.02
IV	1000	5.40	6.37	5.89	6.18
		999.59	998.30	996.10 *	998.19 *
V	2500	6.40	3.61	3.21	2.88
		1403.63	2499.71	2499.51	2499.96
VI	2000	6.38	3.81	3.37	2.69
		1451.96	1996.14	1999.13	1999.2
VII	650	5.95	4.75	5.04	5.04
		649.03	649.94	649.93 *	649.94 *
VIII	1000	5.39	5.32	4.82	4.87
		998.81	999.62	999.09 *	999.82 *
IX	650	4.69	4.67	4.86	4.93
		649.94	649.96	649.95 *	649.92 *
X	300	4.37	4.42	4.45	4.58
		299.66	299.87	299.82	298.93
Average	920 (100.0%)	5.23	4.73	4.56	4.51
		755.20	919.33	919.31	919.45
		(82.1%)	(99.9%)	(99.9%)	(99.9%)

8.5 Summary

This chapter presented the TMA platform, which consists of an assessment life-cycle, a monitoring platform, and measurement instruments (probes) and adaptation services, and it allows defining a trustworthiness level for the applications. TMA enables the self-adaptation of applications based on trustworthiness properties. Quality Models (QMs), which are defined using attributes, weights, operators, and thresholds, are used to portray the trustworthiness level of a cloud application. The attributes reflect the measurement of trustworthiness sub-properties.

We also presented a running example that uses TMA to promote self-adaptation in a cloud application. The managed application is deployed on a Kubernetes cluster, and the performed adaptations scale up and down the number of pods of a server according to resource consumption and performance. The application is tested in an experiment that consists of different workload requests with different request demands (requests/second), and the adaptations are dispatched to meet the expected response time (seconds) and throughput (request responses/second) requirements. We can conclude that TMA allows adaptations based on QMs that measure trustworthiness properties.

TMA can be used either at design-time (while the software is being developed) to identify potential security hotspots (by using SCOLP) or at run-time (during the software execution) to identify trustworthiness problems, such as an intrusion in the system. This is possible due to its flexibility of TMA to interact with the managed application through *probes* and *actuators* and their RESTful interfaces.

Chapter 9

Conclusions and Future Work

The security of software systems is a problem that most enterprises frequently have to deal with. Every day, new vulnerabilities are discovered as a consequence of more software systems being present in our lives and the lack of concern from the developers with security. As of the end of September 2023, more than 20,000 vulnerabilities had been reported according to CVE Details [Özkan, 2023]. New technologies (*e.g.*, Cloud Environments, Edge Computing, Internet of Things (IoT), Large Language Models (LLMs)) are being used, and security mechanisms need to be created to address their security weaknesses.

9.1 Conclusions

The work presented in this thesis advances the state-of-the-art on software security from the vulnerability avoidance perspective. The thesis started by introducing a dataset of software vulnerabilities of five open-source C/C++ projects (Mozilla, Linux Kernel, Xen, Apache httpd, and glibc) enhanced with static information, namely SMs from the source code and alerts reported by SATs. Such vulnerabilities are organized by categories, which correspond to the lack or improper use of one of the OWASP best practices for secure development. The static information is extracted from both the vulnerable and neutral (non-vulnerable) versions, and this allows the work that we performed afterward.

An analysis of the SAT alerts in one of the C/C++ projects (Mozilla) was presented to better understand the use of static analysis in large projects. The large size of the project results in a considerable number of reported alerts, making it difficult for the development team to analyze them all. Also, none of the analyzed SATs (CppCheck and Flawfinder) presented good metrics (precision, recall, or f-measure) for the reported vulnerabilities. The SATs were also analyzed considering different vulnerability categories, but the results are not good as well, although CppCheck performs slightly better than Flawfinder.

The fixes of buffer overflow vulnerabilities were studied to understand how developers usually correct them. The vulnerabilities of three projects of the dataset (Linux Kernel, Mozilla, and Xen) are classified using the Orthogonal Defect Clas-

sification (ODC). Most vulnerability fixes involve a checking (*e.g.*, an if-clause) that is either missing or incorrect in the source code, which is difficult to be detected by the analyzed SATs. Furthermore, fixes usually add more code to the codebase, which results in an increase in the SMs related to volume and complexity.

We demonstrated the use of ML to detect vulnerable code units in two different studies. In the first one, SMs and SAT alerts are used as input (features) to classical ML algorithms (DT, RF, XGB, and bagging) to detect vulnerable files. Although it was possible to obtain good values for precision and recall, it was not possible to obtain them in the same configuration with the same hyperparameters. Bagging and XGB are the ones that performed better. The second study used a deep learning approach (with DGCNN and VGG) to detect vulnerable functions. The CFGs of each function are used as input for the network, and the features are the SMs and attributes that we defined related to memory management. The results were similar to the ones obtained with the classical ML algorithms.

Based on the results obtained with the ML algorithms, we decided to characterize the code units instead of detecting vulnerable ones. Hence, we developed SCOLP, an LSP-based approach to categorize the code units and guide the software development team analysis. SCOLP uses SMs and memory management-related attributes as input of QMs, which outputs a score to each code unit. This score is used to assign each code unit to priority groups. We demonstrated its use with code units (functions) of the Linux Kernel project. A validation was performed with security experts, showing that SCOLP results are comparable with the assignment of security experts.

The last contribution of this thesis is the TMA platform, which allows adding self-adaptation capabilities to applications. As it also uses QMs, it can be integrated with SCOLP. Due to its self-adaptation abilities, it allows adaptations during run-time, *i.e.*, while the software is in execution. It could also be used to detect security issues, such as an intrusion. We demonstrated TMA in the context of a Kubernetes application, which has its pods (components) scaled up and down through the QMs and adaptation plans created by the TMA platform.

9.2 Future Work

Based on the results and conclusions from our work, we propose several research directions for the future:

- **Expand the vulnerability dataset:** the process presented in this thesis is generic, and it can be easily expanded to add other projects regardless of the programming languages, as long as their vulnerabilities are reported on CVE Details and the source code is hosted in a Git-based repository. Also, due to the structure of the database, it allows adding other SMs and alerts of different SATs. The vulnerability collection mechanism should also be

automated to allow a continuous checking of CVE Details to obtain new vulnerabilities and update the already collected vulnerabilities. Hence, automation and a more frequent execution should be performed to have the most recent disclosed vulnerabilities.

- **Use ODC to analyze other vulnerabilities beyond buffer overflow:** the analysis of the vulnerability fixes was performed only to buffer overflow vulnerabilities. Other vulnerability types would also benefit from an ODC with the support of SMs and SAT alerts. The findings could further support the development of SATs and other vulnerability detection techniques.
- **Historical data to predict vulnerable code units:** the vulnerabilities and attacks evolve over time. Hence, it is important to predict potentially vulnerable code units based on the previously reported vulnerabilities. Also, more recent vulnerability data should have a larger weight than the oldest vulnerability data. This may help predicting vulnerabilities more accurately.
- **Expand the use of deep learning to detect vulnerable code units:** the work presented uses only one project (Linux Kernel). However, other projects should be used besides adding attributes tailored to other vulnerability types, and not only attributes related to memory-management. Other deep learning techniques may also be explored.
- **Expand and create other QMs for SCOLP:** the created QMs are only for function code level. Other QMs need to be created for this level, as well as for other code levels (*i.e.*, files and classes). Also, a visual tool may be created to validate the QMs structure and weights, allowing the visualization of the scores and the assignment to the priority groups. The number of priority groups can also be varied to obtain a fine-grained assignment of the code units in the priority groups.
- **Apply SCOLP to other projects:** not only C/C++ projects but also projects of other programming languages can be categorized using SCOLP. This would allow understanding the characteristics of such projects, as well as making the use of SCOLP not limited to a single project of C/C++.
- **Integrate SCOLP in a CI tool:** the categorization provided by SCOLP can be integrated in a CI tool (*e.g.*, Jenkins, Microsoft AzureDevOps). This would allow the software development teams to understand the categorization of the code units when a new commit happens in the source code as part of a DevSecOps methodology. As soon as a code unit is assigned to a potential critical priority group, the development team can start addressing the issue.
- **Integrate dynamic data to support the security characterization:** all the work presented in this thesis focuses on static information obtained from the source code (static data). However, data obtained from execution may also reveal information about the software system utilization. Such data can be analyzed and integrated into mechanisms such as SCOLP and TMA to improve software vulnerability detection and characterization.

References

- Acunetix. Acunetix. <https://www.acunetix.com>, 2019. Accessed: 2019-05-30.
- Anas Al-Far, Abdallah Qusef, and Sufyan Almajali. Measuring impact score on confidentiality, integrity, and availability using code metrics. In *2018 International Arab Conference on Information Technology (ACIT)*, pages 1–9, 2018. doi: 10.1109/ACIT.2018.8672678.
- Areej Algaith, Paulo Jorge Costa Nunes, F. D. Rivera San Jose, Ilir Gashi, and Marco Vieira. Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools. *2018 14th European Dependable Computing Conference (EDCC)*, pages 57–64, 2018.
- Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2014.
- H. Alves, B. Fonseca, and N. Antunes. Software metrics and security vulnerabilities: Dataset and exploratory study. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 37–44, Sep. 2016a. doi: 10.1109/EDCC.2016.34.
- H. Alves, B. Fonseca, and N. Antunes. Experimenting Machine Learning Techniques to Predict Vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156, Oct 2016b. doi: 10.1109/LADC.2016.32.
- Amazon. AWS Auto Scaling, 2009. URL <https://aws.amazon.com/autoscaling/>. Accessed: 2018-12-01.
- Amr Amin, Amgad Eldessouki, Menna Tullah Magdy, Nouran Abdeen, Hanan Hindy, and Islam Hegazy. Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach. *Information*, 10(10), 2019. ISSN 2078-2489. doi: 10.3390/info10100326. URL <https://www.mdpi.com/2078-2489/10/10/326>.
- David J Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.
- N. Antunes and M. Vieira. Detecting SQL Injection Vulnerabilities in Web Services. In *2009 Fourth Latin-American Symposium on Dependable Computing*, pages 17–24, Sep. 2009a. doi: 10.1109/LADC.2009.21.
- N. Antunes and M. Vieira. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in

- Web Services. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 301–306, Nov 2009b. doi: 10.1109/PRDC.2009.54.
- N. Antunes and M. Vieira. Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples. *IEEE Transactions on Services Computing*, 8(2):269–283, 2015. ISSN 1939-1374. doi: 10.1109/TSC.2014.2310221.
- Apache. Apache Mesos, 2009. URL <http://mesos.apache.org>. Accessed: 2018-11-29.
- Apache. Apache kafka, 2011. URL <https://kafka.apache.org>. Accessed: 2019-02-08.
- B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security Privacy*, 3(1):84–87, 2005. doi: 10.1109/MSP.2005.23.
- A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. 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)*, pages 161–168, Sep. 2017. doi: 10.1109/SYNASC.2017.00035.
- A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106, Sep. 2011. doi: 10.1109/ESEM.2011.18.
- Andrew Austin, Casper Holmgreen, and Laurie Williams. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. *Information and Software Technology*, 55(7):1279 – 1288, 2013. ISSN 0950-5849.
- A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2.
- N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, Sep. 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.130.
- D. Balzarotti, M. Cova, V. Felmetzger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401, May 2008. doi: 10.1109/SP.2008.22.
- Craig Beaman, Michael Redbourne, J. Darren Mummery, and Saqib Hakak. Fuzzing vulnerability discovery techniques: Survey, challenges and future directions. *Computers & Security*, 120:102813, 2022. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2022.102813>. URL <https://www.sciencedirect.com/science/article/pii/S0167404822002073>.

- Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. The agile manifesto, 2001.
- S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 427–430, March 2011. doi: 10.1109/ICST.2011.48.
- BeyondTrust. Microsoft Vulnerabilities Report 2020, 2020. URL <https://www.beyondtrust.com/resources/whitepapers/microsoft-vulnerability-report-2020>. Accessed: 2023-06-05.
- Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020. doi: 10.1109/ACCESS.2020.3016774.
- Alexandre Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. Understanding how to use static analysis tools for detecting cryptography misuse in software. *IEEE Transactions on Reliability*, 68(4):1384–1403, 2019. doi: 10.1109/TR.2019.2937214.
- Chuck Brooks. Cybersecurity Trends & Statistics For 2023; What You Need To Know, 2023. URL <https://www.forbes.com/sites/chuckbrooks/2023/03/05/cybersecurity-trends--statistics-for-2023-more-treachery-and-risk-ahead-as-attack-surface-and-hacker-capabilities-grow/?sh=7f49265619db>. Accessed: 2023-04-24.
- Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. Dynamic management of virtual infrastructures. *Journal of Grid Computing*, 13(1): 53–70, Mar 2015. ISSN 1572-9184. doi: 10.1007/s10723-014-9296-5.
- Gui-lin Cai, Bao-sheng Wang, Wei Hu, and Tian-zuo Wang. Moving target defense: state of the art and characteristics. *Frontiers of Information Technology & Electronic Engineering*, 17(11):1122–1153, 2016.
- G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., USA, 1st edition, 2013. ISBN 1617290955.
- Sicong Cao, Xiaobing Sun, Lili Bo, Ying Wei, and Bin Li. Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection. *Information and Software Technology*, 136:106576, 2021. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2021.106576>. URL <https://www.sciencedirect.com/science/article/pii/S0950584921000586>.
- CastSoftware. Cast - Software Intelligence for Digital Leaders. <https://www.castsoftware.com/>, 1990. Accessed: 2019-05-30.
- Cagatay Catal and Banu Diri. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8):1040–1058, 2009. ISSN 0020-0255. doi: <https://doi.org/10.1016/j.ins.2009.05.011>.

//doi.org/10.1016/j.ins.2008.12.001. URL <https://www.sciencedirect.com/science/article/pii/S0020025508005173>.

CERT. The CERT Division. <https://www.sei.cmu.edu/about/divisions/cert/>, 1988. Accessed: 2019-03-15.

Checkstyle. Checkstyle. <http://checkstyle.sourceforge.net/>, 2001. Accessed: 2019-05-30.

Xiang Chen, Yingquan Zhao, Zhanqi Cui, Guozhu Meng, Yang Liu, and Zan Wang. Large-scale empirical studies on effort-aware security vulnerability prediction methods. *IEEE Transactions on Reliability*, 69(1):70–87, 2020. doi: 10.1109/TR.2019.2924932.

B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6): 76–79, Nov 2004. ISSN 1540-7993. doi: 10.1109/MSP.2004.111.

Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321424778.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. ISSN 0098-5589. doi: 10.1109/32.295895.

Shyam R. Chidamber and Chris F. Kemerer. Towards a Metrics Suite for Object Oriented Design. *SIGPLAN Not.*, 26(11):197–211, November 1991. ISSN 0362-1340. doi: 10.1145/118014.117970.

R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. . Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992. doi: 10.1109/32.177364.

Jin-Hee Cho, Dilli P. Sharma, Hooman Alavizadeh, Seunghyun Yoon, Noam Ben-Asher, Terrence J. Moore, Dong Seong Kim, Hyuk Lim, and Frederica F. Nelson. Toward proactive, adaptive defense: A survey on moving target defense. *IEEE Communications Surveys & Tutorials*, 22(1):709–745, 2020. doi: 10.1109/COMST.2019.2963791.

Istehad Chowdhury and Mohammad Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 1963–1969, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi: 10.1145/1774088.1774504. URL <http://doi.acm.org/10.1145/1774088.1774504>.

CISA. APT Actors Chaining Vulnerabilities Against SLTT, Critical Infrastructure, and Elections Organizations, 2020. URL <https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-283a>. Accessed: 2023-06-05.

Google Cloud. State of DevOps Report 2022, 2022. URL <https://cloud.google.com/devops/state-of-devops>. Accessed: 2023-07-10.

- Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.
- José D’Abruzzo Pereira, Rui Silva, Nuno Antunes, Jorge L. M. Silva, Breno de França, Regina Moraes, and Marco Vieira. A platform to enable self-adaptive cloud applications using trustworthiness properties. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS ’20*, page 71–77, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379625. doi: 10.1145/3387939.3391608. URL <https://doi.org/10.1145/3387939.3391608>.
- José D’Abruzzo Pereira, Nuno Lourenço, and Marco Vieira. On the Use of Deep Graph CNN to Detect Vulnerable C Functions. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing, LADC ’22*, page 45–50, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450397377. doi: 10.1145/3569902.3569913. URL <https://doi.org/10.1145/3569902.3569913>.
- José D’Abruzzo Pereira and Marco Vieira. On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 97–102, 2020. doi: 10.1109/EDCC51268.2020.00025.
- José D’Abruzzo Pereira and Marco Vieira. An approach to characterize the security of open-source functions using lsp. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 137–147, 2023. doi: 10.1109/ISSRE59848.2023.00073.
- José D’Abruzzo Pereira, João R. Campos, and Marco Vieira. Machine learning to combine static analysis alerts with software metrics to detect security vulnerabilities: An empirical study. In *2021 17th European Dependable Computing Conference (EDCC)*, pages 1–8, 2021. doi: 10.1109/EDCC53658.2021.00008.
- José D’Abruzzo Pereira, João Henggeler Antunes, and Marco Vieira. A Software Vulnerability Dataset of Large Open Source C/C++ Projects. In *2022 IEEE 27th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 152–163, 2022. doi: 10.1109/PRDC55274.2022.00029.
- Johannes Dahse and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *In Symposium on Network and Distributed System Security (NDSS)*, pages 23–26, 01 2014. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23262.
- Maryam Davari and Mohammad Zulkernine. Analysing vulnerability reproducibility for firefox browser. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 674–681, 2016. doi: 10.1109/PST.2016.7906955.
- Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs,

- Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaella Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, Lecture Notes in Computer Science, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-35813-5. doi: 10.1007/978-3-642-35813-5_1.
- digital.ai. 16th State of Agile Report, 2022. URL <https://digital.ai/resource-center/analyst-reports/state-of-agile-report/>. Accessed: 2023-07-10.
- Docker. Swarm Mode Overview, 2017. URL <https://docs.docker.com/engine/swarm/>. Accessed: 2018-11-29.
- Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*, pages 18–24. Pearson Education, 2006.
- Jozo Dujmovic. *Soft computing evaluation logic: The LSP decision method and its applications*. John Wiley & Sons, 2018.
- Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
- José D’Abruzzo Pereira, Naghmeh Ivaki, and Marco Vieira. Characterizing buffer overflow vulnerabilities in large c/c++ projects. *IEEE Access*, 9:142879–142892, 2021. doi: 10.1109/ACCESS.2021.3120349.
- Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016. doi: 10.1109/MS.2016.68.
- EscherTechnologies. Escher Technologies. <http://eschertech.com/>, 1997. Accessed: 2019-05-30.
- Facebook. Infer static analyzer. <https://fbinfer.com>, 2013. Accessed: 2022-05-21.
- M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. ISSN 0018-8670. doi: 10.1147/sj.153.0182.
- Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR ’20*, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375177. doi: 10.1145/3379597.3387501. URL <https://doi.org/10.1145/3379597.3387501>.

- C. Ferri, J. Hernández-Orallo, and R. Modroiu. An experimental comparison of performance measures for classification. *Pattern Recognition Letters*, 30(1):27–38, 2009. ISSN 0167-8655. doi: <https://doi.org/10.1016/j.patrec.2008.08.010>. URL <https://www.sciencedirect.com/science/article/pii/S0167865508002687>.
- Katarzyna Filus and Joanna Domańska. Software vulnerabilities in tensorflow-based deep learning applications. *Computers & Security*, 124:102948, 2023. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2022.102948>. URL <https://www.sciencedirect.com/science/article/pii/S0167404822003406>.
- First. Common Vulnerability Scoring System SIG. <https://www.first.org/cvss/>, 1995. Accessed: 2019-04-13.
- FIRST. FIRST - Improving Security Together, 2005. URL <https://www.first.org>. Accessed: 2023-10-28.
- FIRST. Common Vulnerability Scoring System v3.1: Specification Document. <https://www.first.org/cvss/v3.1/specification-document>, 2019. Accessed: 2023-11-07.
- Felix Fischer, Yannick Stachelscheid, and Jens Grossklags. The effect of google search on software security: Unobtrusive security interventions via content re-ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21*, page 3070–3084, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384544. doi: 10.1145/3460120.3484763. URL <https://doi.org/10.1145/3460120.3484763>.
- Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies, SQUADE '18*, pages 13–20, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5737-1. doi: 10.1145/3194095.3194100. URL <http://doi.acm.org/10.1145/3194095.3194100>.
- The Open Web Application Security Project Foundation. Welcome to the OWASP Top 10 - 2021. <https://owasp.org/www-project-top-ten/2017/>, 2021. Accessed: 2023-10-22.
- Martin Fowler. Codesmell. <https://martinfowler.com/bliki/CodeSmell.html>, 2006. Accessed: 2019-05-29.
- Willy Ronald Jimenez Freitez, Amel Mammar, and Ana Rosa Cavalli. Software vulnerabilities, prevention and detection methods: a review. In *SEC-MDA 2009: Security in Model Driven Architecture*, pages 1–11, 2009.
- J. E. Gaffney. Metrics in software quality assurance. In *Proceedings of the ACM '81 Conference, ACM '81*, page 126–130, New York, NY, USA, 1981. Association for Computing Machinery. ISBN 0897910494. doi: 10.1145/800175.809854. URL <https://doi.org/10.1145/800175.809854>.

David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure Rainbow: Architecture- Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004. doi: 10.1109/MC.2004.175.

Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50(4), aug 2017. ISSN 0360-0300. doi: 10.1145/3092566. URL <https://doi.org/10.1145/3092566>.

Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems — survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007. ISSN 0167-9236. doi: <https://doi.org/10.1016/j.dss.2006.06.011>. URL <https://www.sciencedirect.com/science/article/pii/S0167923606000807>. Decision Support Systems in Emerging Economies.

Google. Kubernetes Horizontal Pod Autoscaler, 2014a. URL <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/>. Accessed: 2018-12-01.

Google. Kubernetes - production-grade container orchestration, 2014b. URL <https://kubernetes.io>. Accessed: 2018-11-29.

Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi, editors, *Intelligent Computing and Applications*, pages 581–591. Springer India, 2015. ISBN 978-81-322-2268-2.

GrammaTech. GrammaTech - Software Assurance and Cyber-Security Solutions. <https://www.grammatech.com/>, 2018. Accessed: 2019-05-30.

The Guardian. Recently uncovered software flaw ‘most critical vulnerability of the last decade’, 2021. URL <https://www.theguardian.com/technology/2021/dec/10/software-flaw-most-critical-vulnerability-log-4-shell>. Accessed: 2023-06-20.

Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., August 2013. USENIX Association. ISBN 978-1-931971-03-4. URL <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller>.

Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

Hazim Hanif, Mohd Hairul Nizam 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. *Journal of Network and Computer Applications*, page 103009, 2021.

- Nima Shiri Harzevili, Jiho Shin, Junjie Wang, Song Wang, and Nachiappan Nagappan. Automatic static bug detection for machine learning libraries: Are we there yet? In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 795–806, 2023. doi: 10.1109/ISSRE59848.2023.00042.
- David Hauzar and Jan Kofron. Framework for Static Analysis of PHP Applications. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. doi: 10.4230/LIPIcs.ECOOP.2015.689.
- Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363 – 387, 2011. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2010.12.007>. URL <http://www.sciencedirect.com/science/article/pii/S0950584910002235>. Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.
- S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981. doi: 10.1109/TSE.1981.231113.
- Seah Higgins. Finding and Fixing C++ Vulnerabilities, 2020. URL <https://www.securecoding.com/blog/finding-and-fixing-c-vulnerabilities/>. Accessed: 2023-05-04.
- J.R. Horgan, S. London, and M.R. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994. doi: 10.1109/2.312032.
- IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, 31: 1–6, 2006.
- IBM. IBM AppScan. <https://www.ibm.com/security/application-security/appscan>, 2019. Accessed: 2019-05-30.
- N. Imtiaz, A. Rahman, E. Farhana, and L. Williams. Challenges with Responding to Static Analysis Tool Alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249, 2019.
- Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333, 2019. doi: 10.1109/ISSRE.2019.00040.
- Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards efficient heap overflow discovery. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 989–1006, 2017.
- Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, PROMISE '08*,

- page 11–18, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580364. doi: 10.1145/1370788.1370793. URL <https://doi.org/10.1145/1370788.1370793>.
- JLint. JLint. <http://jlint.sourceforge.net>, 2002. Accessed: 2019-05-30.
- M. Johns, B. Engelmann, and J. Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 335–344, Dec 2008. doi: 10.1109/ACSAC.2008.36.
- Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.
- N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 6 pp.–263, May 2006. doi: 10.1109/SP.2006.29.
- Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070521.
- Seokmo Kim, R Young Chul Kim, and Young B Park. Software vulnerability detection methodology combined with static and dynamic analysis. *Wireless Personal Communications*, 89:777–793, 2016.
- Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. ISA: A Source Code Static Vulnerability Detection System Based on Data Fusion. In *Proceedings of the 2Nd International Conference on Scalable Information Systems, InfoScale '07*, pages 55:1–55:7, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-1-59593-757-5. URL <http://dl.acm.org/citation.cfm?id=1366804.1366875>.
- Kendra Kratkiewicz and Richard Lippmann. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. In *Proc. of Workshop on the Evaluation of Software Defect Detection Tools*, page 19, 2005.
- N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, pages 230–239, June 1998. doi: 10.1109/FTCS.1998.689474.
- Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17(PB):184–206, February 2015. ISSN 15741192. doi: 10.1016/j.pmcj.2014.09.009.

- Zachary Kurtz. The Vectors of Code: On Machine Learning for Software. https://insights.sei.cmu.edu/sei_blog/2019/06/vectors-of-code-on-the-foundations-of-machine-learning-for-software.html, 2019. Accessed: 2019-06-11.
- J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.
- Triet H. M. Le, Huaming Chen, and M. Ali Babar. A Survey on Data-Driven Software Vulnerability Assessment and Prioritization. *ACM Comput. Surv.*, 55(5), dec 2022. ISSN 0360-0300. doi: 10.1145/3529757. URL <https://doi.org/10.1145/3529757>.
- Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.*, 52(6), nov 2019. ISSN 0360-0300. doi: 10.1145/3359981. URL <https://doi.org/10.1145/3359981>.
- James Lewis and Martin Fowler. *Microservices*, 2014. URL <https://martinfowler.com/articles/microservices.html>. Accessed: 2018-11-09.
- Zhen 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 *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018. doi: 10.14722/ndss.2018.23158. URL <https://doi.org/10.14722%2Fndss.2018.23158>.
- Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2022. doi: 10.1109/TDSC.2021.3051525.
- B. Liu, L. Shi, Z. Cai, and M. Li. Software Vulnerability Discovery Techniques: A Survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 152–156. IEEE, Nov 2012. doi: 10.1109/MINES.2012.202.
- Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 1547–1559, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380923. URL <https://doi.org/10.1145/3377811.3380923>.
- Qixu Liu and Yuqing Zhang. Vrss: A new system for rating and scoring vulnerabilities. *Computer Communications*, 34(3):264–273, 2011. ISSN 0140-3664. doi: <https://doi.org/10.1016/j.comcom.2010.04.006>. URL <https://www.sciencedirect.com/science/article/pii/S014036641000174X>. Special Issue of Computer Communications on Information and Future Communication Security.

- Fábio Lopes, João Agnelo, César A. Teixeira, Nuno Laranjeiro, and Jorge Bernardino. Automating orthogonal defect classification using machine learning algorithms. *Future Generation Computer Systems*, 102:932–947, 2020. ISSN 0167-739X. doi: <https://doi.org/10.1016/j.future.2019.09.009>. URL <https://www.sciencedirect.com/science/article/pii/S0167739X19308283>.
- P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, July 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.114.
- Frank D. Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267 – 7279, 2013. ISSN 0957-4174. doi: <https://doi.org/10.1016/j.eswa.2013.07.033>.
- P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.60.
- G. V. Marconato, V. Nicomette, and M. Kaâniche. Security-related vulnerability life cycle analysis. In *2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–8, Oct 2012. doi: 10.1109/CRiSIS.2012.6378954.
- Daniel Marjamäki. Cppcheck - A tool for static C/C++ code analysis, 2007. URL <http://cppcheck.sourceforge.net>. Accessed: 2019-08-30.
- J.P. Marques de Sá. *Pattern Recognition*. Springer-Verlag Berlin Heidelberg, 1 edition, 2001. ISBN 9783642566516.
- Miquel Martínez, Juan-Carlos Ruiz, Nuno Antunes, David de Andrés, and Marco Vieira. A multi-criteria analysis of benchmark results with expert support for security tools. *IEEE Transactions on Dependable and Secure Computing*, 19(4): 2151–2164, 2022. doi: 10.1109/TDSC.2020.3048202.
- Mark Maunder. Panama Papers: Email Hackable via WordPress, Docs Hackable via Drupal, 2016. URL <https://www.wordfence.com/blog/2016/04/panama-papers-wordpress-email-connection/>. Accessed: 2023-06-22.
- T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.
- Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN 0321356705.
- Nancy Mead, Eric Hough, and Ted Stehney II. Security Quality Requirements Engineering Technical Report. Technical Report CMU/SEI-2005-TR-009, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. URL <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7657>.
- Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives. In *Proceedings of the 23rd International Conference on World Wide*

- Web*, WWW '14, pages 63–74, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2744-2. doi: 10.1145/2566486.2568024.
- Ibéria Medeiros, Nuno Neves, and Miguel Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, page 1–11, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931041. URL <https://doi.org/10.1145/2931037.2931041>.
- N. Medeiros, N. Ivaki, P. Costa, and M. Vieira. Software Metrics as Indicators of Security Vulnerabilities. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, Oct 2017. doi: 10.1109/ISSRE.2017.11.
- N. Medeiros, N. Ivaki, P. Costa, and M. Vieira. Vulnerable Code Detection Using Software Metrics and Machine Learning. *IEEE Access*, 8:219174–219198, 2020. doi: 10.1109/ACCESS.2020.3041181.
- Nadia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. Trustworthiness models to categorize and prioritize code for security improvement. *Journal of Systems and Software*, 198:111621, 2023. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2023.111621>. URL <https://www.sciencedirect.com/science/article/pii/S016412122300016X>.
- Nádia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. An Approach for Trustworthiness Benchmarking Using Software Metrics. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 84–93, 2018. doi: 10.1109/PRDC.2018.00019.
- Syaeful Karim Meiliana, Harco Leslie Hendric Spits Warnars, Ford Lumban Gaol, Edi Abdurachman, and Benfano Soewito. Software metrics for fault prediction using machine learning approaches: A literature review with promise repository dataset. In *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, pages 19–23, 2017. doi: 10.1109/CYBERNETICSCOM.2017.8311708.
- Peter Mell, Jonathan Spring, Dave Dugal, Srividya Ananthakrishna, Francesco Casotto, Troy Fridley, Christopher Ganas, Arkadeep Kundu, Phillip Nordwall, Vijayamurugan Pushpanathan, et al. Measuring the common vulnerability scoring system base score equation. *National Institute of Standards and Technology, Gaithersburg, MD*, 2022.
- N. Meng, Q. Wang, Q. Wu, and H. Mei. An Approach to Merge Results of Multiple Static Analysis Tools (Short Paper). In *2008 The Eighth International Conference on Quality Software*, pages 169–174, Aug 2008. doi: 10.1109/QSIC.2008.30.
- Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL <https://arxiv.org/abs/1301.3781>.

- Aleksandar Milenkoski, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Bryan D. Payne. Evaluating computer intrusion detection systems: A survey of common practices. *ACM Comput. Surv.*, 48(1), sep 2015. ISSN 0360-0300. doi: 10.1145/2808691. URL <https://doi.org/10.1145/2808691>.
- MITRE. Common Weakness Enumeration - A Community-Developed List of Software Weakness Types. <https://cwe.mitre.org>, 2006a. URL <https://cwe.mitre.org>. Accessed: 2021-05-03.
- MITRE. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. <https://cwe.mitre.org/data/definitions/119.html>, 2006b. Accessed: 2021-05-03.
- MITRE. CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). <https://cwe.mitre.org/data/definitions/120.html>, 2006c. Accessed: 2023-11-04.
- MITRE. CWE-20: Improper Input Validation. <https://cwe.mitre.org/data/definitions/20.html>, 2006d. Accessed: 2022-01-16.
- MITRE. CWE-399: Resource Management Errors. <https://cwe.mitre.org/data/definitions/399.html>, 2006e. Accessed: 2022-01-16.
- MITRE. CWE-416: Use After Free. <https://cwe.mitre.org/data/definitions/416.html>, 2006f. Accessed: 2022-05-19.
- MITRE. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, 2021. URL https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. Accessed: 2023-11-10.
- Nazila Mohammadi, Sachar Paulus, Mohamed Bishr, Andreas Metzger, Holger Könnecke, Sandro Hartenstein, and Klaus Pohl. An analysis of software quality attributes and their contribution to trustworthiness. In *CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science, Aachen, Germany, 8-10 May, 2013*, pages 542–552. SciTePress, 2013.
- Susan Moore. 7 top trends in cybersecurity for 2022, 2022. URL <https://www.gartner.com/en/articles/7-top-trends-in-cybersecurity-for-2022>.
- Susan Moore and Emma Keen. Gartner Forecasts Worldwide Information Security Spending to Exceed \$124 Billion in 2019. <https://www.gartner.com/en/newsroom/press-releases/2018-08-15-gartner-forecasts-worldwide-information-security-spending-to-exceed-124-billion-in-2019>, 2018. Accessed: 2019-06-03.
- Patrick J Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*, 23(3):1383–1421, 2018.
- Haralambos Mouratidis, Paolo Giorgini, and Gordon Manson. When security meets software engineering: a case of modelling secure information systems.

- Information Systems*, 30(8):609–629, 2005. ISSN 0306-4379. doi: <https://doi.org/10.1016/j.is.2004.06.002>. URL <https://www.sciencedirect.com/science/article/pii/S0306437904000626>.
- Ian Muscat. Cyber Threats vs Vulnerabilities vs Risks. <https://www.acunetix.com/blog/articles/cyber-threats-vulnerabilities-risks/>, 2017. Accessed: 2019-06-20.
- T. Muske and A. Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, Oct 2016. doi: 10.1109/SCAM.2016.25.
- Tukaram Muske and Uday P. Khedker. Efficient elimination of false positives using static analysis. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 270–280, 2015. doi: 10.1109/ISSRE.2015.7381820.
- Laxman Muthiyah. Hacking Facebook Pages, 2015. URL <https://thezerohack.com/hacking-facebook-pages>. Accessed: 2023-06-20.
- Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083143. URL <http://doi.acm.org/10.1145/1082983.1083143>.
- A. A. Neto and M. Vieira. Trustworthiness Benchmarking of Web Applications Using Static Code Analysis. In *2011 Sixth International Conference on Availability, Reliability and Security*, pages 224–229, Aug 2011. doi: 10.1109/ARES.2011.37.
- N. Neves, J. Antunes, M. Correia, P. Verissimo, and R. Neves. Using attack injection to discover new vulnerabilities. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 457–466, 2006.
- Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology*, page 106614, 2021.
- P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira. On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study. In *2017 13th European Dependable Computing Conference (EDCC)*, pages 121–128, Sep. 2017. doi: 10.1109/EDCC.2017.16.
- P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira. Benchmarking Static Analysis Tools for Web Security. *IEEE Transactions on Reliability*, 67(3):1159–1175, Sep. 2018. ISSN 0018-9529. doi: 10.1109/TR.2018.2839339.
- P. J. C. Nunes, J. Fonseca, and M. Vieira. phpSAFE: A Security Analysis Tool for OOP Web Application Plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 299–306, June 2015. doi: 10.1109/DSN.2015.16.

- The International Consortium of Investigative Journalists. Giant Leak of Offshore Financial Records Exposes Global Array of Crime and Corruption, 2016. URL <https://www.occrp.org/en/panamapapers/overview/intro/>. Accessed: 2023-06-22.
- National Institute of Standards and Technology (NIST). National Vulnerability Database, 2005. URL <https://nvd.nist.gov>. Accessed: 2021-08-24.
- National Institute of Standards and Technology. National vulnerability database - vulnerabilities. <https://nvd.nist.gov/vuln>, 2020. Accessed: 2020-11-21.
- Tom Okman. Cybersecurity predictions for 2022, 2022. URL <https://www.forbes.com/sites/forbestechcouncil/2022/03/11/cybersecurity-predictions-for-2022/?sh=22aca7257749>.
- C. Pahl and B. Lee. Containers and clusters for edge cloud architectures – a technology review. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 379–386, Aug 2015. doi: 10.1109/FiCloud.2015.35.
- Y. Pang, X. Xue, and A. S. 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)*, pages 543–548, Dec 2015. doi: 10.1109/ICMLA.2015.99.
- Parasoft. Parasoft cppTest - C/C++ Static Code Analysis, 2010. URL <https://www.parasoft.com/products/parasoft-c-ctest/c-c-static-analysis/>. Accessed: 2021-05-11.
- Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.
- Xutan Peng, Yipeng Zhangira, Jingfeng Yang, and Mark Stevenson. On the vulnerabilities of text-to-sql models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–12, 2023. doi: 10.1109/ISSRE59848.2023.00047.
- Sten Pittet. Learn Continuous Deployment with Bitbucket Pipelines, 2023. URL <https://www.atlassian.com/devops/continuous-delivery-tutorials/continuous-deployment-tutorial>. Accessed: 2023-04-24.
- PMD. PMD - An extensible cross-language static code analyzer. <https://pmd.github.io/>, 2004. Accessed: 2019-05-30.
- MC Prasad, Lilly Florence, and Arti Arya. A study on software metrics based software defect prediction using data mining and machine learning techniques. *International Journal of Database Theory and Application*, 8(3):179–190, 2015.
- Luís Prates, João Faustino, Miguel Silva, and Rúben Pereira. DevSecOps Metrics. In Stanisław Wrycza and Jacek Maślankowski, editors, *Information Systems: Research, Development, Applications, Education*, pages 77–90, Cham, 2019. Springer International Publishing. ISBN 978-3-030-29608-7.

- Rahul Premraj and Kim Herzig. Network versus code metrics to predict defects: A replication study. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 215–224, 2011. doi: 10.1109/ESEM.2011.30.
- João R. Campos, Marco Vieira, and Ernesto Costa. Prophetus: Machine Learning Framework for the Development of Predictive Models for Reliable and Secure Software. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Oct 2019.
- RedHat. Drools - Business Rules Management System, 2011. URL www.drools.org. Accessed: 2020-01-20.
- RedHat. Wildfly, 2014. URL <http://www.wildfly.org/>. Accessed: 2019-02-14.
- Sofia Reis and Rui Abreu. Secbench: A database of real security vulnerabilities. In *SecSE@ ESORICS*, pages 69–85, 2017.
- Syed Rizvi, RJ Orr, Austin Cox, Prithvee Ashokkumar, and Mohammad R. Rizvi. Identifying the attack surface for iot network. *Internet of Things*, 9:100162, 2020. ISSN 2542-6605. doi: <https://doi.org/10.1016/j.iot.2020.100162>. URL <https://www.sciencedirect.com/science/article/pii/S2542660520300056>.
- Armin Ronacher. Flask - Web development, one drop at a time, 2010. URL <http://flask.pocoo.org/>. Accessed: 2019-02-08.
- Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *CoRR*, abs/1807.04320, 2018.
- F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *2008 Third International Conference on Systems and Networks Communications*, pages 23–26, Oct 2008. doi: 10.1109/ICSNC.2008.44.
- SAMATE. Welcome to the Software Assurance Metrics And Tool Evaluation (SAMATE) Website!, 2005. URL <https://www.nist.gov/itl/ssd/software-quality-group/samate>. Accessed: 2023-07-05.
- Luciano Sampaio and Alessandro Garcia. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software*, 113: 337–361, 2016. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2015.12.021>. URL <http://www.sciencedirect.com/science/article/pii/S0164121215002873>.
- Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Security as Culture: A Systematic Literature Review of DevSecOps. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20*, page 266–269, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379632. doi: 10.1145/3387940.3392233. URL <https://doi.org/10.1145/3387940.3392233>.

- Joint Technical Committee ISO/IEC JTC1. Subcommittee SC27. ISO/IEC 27000:2018 - Information technology — Security techniques — Information security management systems — Overview and vocabulary. Standard, International Organization for Standardization, Geneva, CH, February 2018.
- R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 451–460, Nov 2013. doi: 10.1109/ISSRE.2013.6698898.
- Ken Schwaber and Mike Beedle. *Agile software development with Scrum*. Prentice Hall PTR, 2001.
- Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010. doi: 10.1109/SP.2010.26.
- SciTools. SciTools Understand - Metrics, 2011. URL <https://scitools.com/feature/metrics/>. Accessed: 2020-04-03.
- Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. doi: 10.1109/ACCESS.2017.2685629.
- Abhishek Sharma, Sangeeta Sabharwal, and Sushama Nagpal. A hybrid scoring system for prioritization of software vulnerabilities. *Computers & Security*, 129: 103256, 2023. ISSN 0167-4048. doi: <https://doi.org/10.1016/j.cose.2023.103256>. URL <https://www.sciencedirect.com/science/article/pii/S0167404823001669>.
- V.Y. Shen, S.D. Conte, and H.E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *IEEE Transactions on Software Engineering*, SE-9(2):155–165, 1983. doi: 10.1109/TSE.1983.236460.
- Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011. doi: 10.1109/TSE.2010.81.
- Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.
- Yonghee Shin, Laurie Ann Williams, and Tao Xie. SQLUnitgen: Test case generation for SQL injection detection. Technical report, North Carolina State University. Dept. of Computer Science, 2006.
- S. Shiraishi, V. Mohan, and H. Marimuthu. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 12–15, 2015.

- Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.
- Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014. URL <https://arxiv.org/abs/1409.1556>.
- Rogue Wave Software. Klocwork static code analysis. <https://www.roguewave.com/products-services/klocwork/static-code-analysis>, 2014. Accessed: 2019-05-30.
- Software Engineering Institute – Carnegie Mellon University. SEI CERT C++ Coding Standard, 2006. URL <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>. Accessed: 2021-05-11.
- White Source. What are the most secure programming languages? <https://www.whitesourcesoftware.com/most-secure-programming-languages/>, 2021. Accessed: 2021-03-09.
- Georgios Spanos, Angeliki Sioziou, and Lefteris Angelis. Wivss: A new methodology for scoring information systems vulnerabilities. In *Proceedings of the 17th Panhellenic Conference on Informatics, PCI '13*, page 83–90, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319690. doi: 10.1145/2491845.2491871. URL <https://doi.org/10.1145/2491845.2491871>.
- SpotBugsTeam. SpotBugs, 2017. URL <https://spotbugs.github.io>. Accessed: 2021-05-05.
- Jonathan Spring, Eric Hatleback, A Manion, and D Shic. Towards improving cvss. *Software Engineering Institute, Carnegie Mellon University, Tech. Rep*, 2018.
- Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach. *Journal of Software: Evolution and Process*, 33(3):e2303, 2021.
- Synopsys. Coverity Static Application Security Testing, 2006. URL <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>. Accessed: 2021-05-11.
- Ed Targett. We analysed 90,000+ software vulnerabilities: Here’s what we learned, 2023. URL <https://thestack.technology/analysis-of-cves-in-2022-software-vulnerabilities-cwes-most-dangerous/>. Accessed: 2023-04-24.
- Snyk Security Research Team. Top 5 C++ security risks, 2022. URL <https://snyk.io/blog/top-5-c-security-risks/>. Accessed: 2023-05-04.
- Stephen Thomas, Laurie Williams, and Tao Xie. On automated prepared statement generation to remove sql injection vulnerabilities. *Information and Software Technology*, 51(3):589–598, 2009. ISSN 0950-5849. doi: <https://doi.org/10.1016/j.infsof.2008.08.002>. URL <https://www.sciencedirect.com/science/article/pii/S0950584908001110>.
- TPC. TPC-App (Application Server) Standard Specification, Version 1.3, 2014. URL http://www.tpc.org/tpc_app/. Accessed: 2019-01-23.

- K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: a taxonomy of software security errors. *IEEE Security Privacy*, 3(6):81–84, Nov 2005. ISSN 1558-4046. doi: 10.1109/MSP.2005.159.
- Keith Turpin. OWASP Secure Coding Practices - Quick Reference Guide. https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf, 2010. Accessed: 2019-06-20.
- A. van der Stock, B. Glas, N. Smithline, and T. Gigler. OWASP Top 10 - 2017 - The Ten Most Critical Web Application Security Risks. <https://owasp.org/www-project-top-ten/2017/>, 2017. Accessed: 2021-05-04.
- J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267, Dec 2000. doi: 10.1109/ACSAC.2000.898880.
- Marco Vieira and Nuno Antunes. *Introduction to Software Security Concepts*, pages 29–38. Springer Milan, Milano, 2013. ISBN 978-88-470-2772-5. doi: 10.1007/978-88-470-2772-5_3. URL https://doi.org/10.1007/978-88-470-2772-5_3.
- Tamás Viszok, Péter Hegedűs, and Rudolf Ferenc. Improving vulnerability prediction of javascript functions using process metrics, 2021.
- Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017. ISBN 3319579584, 9783319579580.
- J. Walden, J. Stuckman, and R. Scandariato. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33, Nov 2014. doi: 10.1109/ISSRE.2014.32.
- Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunke. Vudenc: Vulnerability detection with deep learning on a natural codebase for python. *Information and Software Technology*, page 106809, 2022.
- Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, page 307–320, USA, 2006. USENIX Association. ISBN 1931971471.
- David A. Wheeler. Flawfinder, 2001. URL <https://dwheeler.com/flawfinder/>. Accessed: 2019-08-30.
- WPScan. WPScan Vulnerability Database. <https://wpvulndb.com>, 2014. Accessed: 2019-06-13.
- Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. Vulcnn: An image-inspired scalable vulnerability detection system. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2365–2376, 2022.

- Fabian Yamaguchi. Welcome to joern’s documentation! <https://joern.readthedocs.io/en/latest/index.html>, 2014. Accessed: 2022-07-21.
- Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014. doi: 10.1109/SP.2014.44.
- Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–63, 2019. doi: 10.1109/DSN.2019.00020.
- Eric Yuan, Naeem Esfahani, and Sam Malek. A systematic survey of self-protecting software systems. *ACM Trans. Auton. Adapt. Syst.*, 8(4), jan 2014. ISSN 1556-4665. doi: 10.1145/2555611. URL <https://doi.org/10.1145/2555611>.
- Zhen Zeng, Zhun Yang, Dijiang Huang, and Chun-Jen Chung. Licality—likelihood and criticality: Vulnerability risk prioritization through logical reasoning and deep learning. *IEEE Transactions on Network and Service Management*, 19(2):1746–1760, 2022. doi: 10.1109/TNSM.2021.3133811.
- Mengyuan Zhang, Lingyu Wang, Sushil Jajodia, and Anoop Singhal. Network attack surface: Lifting the concept of attack surface to the network level for evaluating networks’ resilience against zero-day attacks. *IEEE Transactions on Dependable and Secure Computing*, 18(1):310–324, 2021. doi: 10.1109/TDSC.2018.2889086.
- Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006. doi: 10.1109/TSE.2006.38.
- Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 111–120, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00020.
- Serkan Özkan. Current cvss score distribution for all vulnerabilities. <https://www.cvedetails.com/cvss-score-distribution.php>, 2020. Accessed: 2021-01-02.
- Serkan Özkan. CVE Details - The ultimate security vulnerability datasource, 2023. URL <https://www.cvedetails.com>. Accessed: 2023-06-07.

Appendix A

ML to Combine Security Vulnerability Alerts from SATs

Security risks, business needs, and industry changes are listed as the top three drivers for security spending according to a survey conducted by Gartner in 2017 [Moore and Keen, 2018]. At the same time, the implementation of the General Data Protection Regulation (GDPR) and other security and privacy regulations are driving change and growth in the security services market [Voigt and Bussche, 2017]. For example, among many other aspects, such regulations require companies to report personal data breaches, otherwise they can get fines that may compromise their revenue. The problem is that software systems are often deployed with vulnerabilities that can open the door for successful attacks.

A software vulnerability is a weakness in a system, and it represents a security risk. When exploited, it may lead to an intrusion [Muscat, 2017] with severe consequences, including financial and data losses. There are many techniques to prevent software vulnerabilities, including best coding practices and vulnerability detection approaches and tools [Liu et al., 2012]. For example, security requirements can be elicited and prioritized using the Security Quality Requirements Engineering (SQUARE) methodology [Mead et al., 2005]; during the application development phase, software developers can use guidelines such as the OWASP secure coding practices to avoid introducing vulnerabilities [Turpin, 2010]; and when the code source is ready, either static (such as SCA) or dynamic (such as penetration testing) approaches can be used.

SCA is one of the most used design-time techniques to detect vulnerabilities, especially through SATs. Many different automated SATs are available in the field (including open-source and commercial ones). However, although practice shows that SATs are typically able to achieve a high vulnerability detection coverage, they usually report a high number of false alarms [Chess and West, 2007], also known as FPs. This happens because the SATs have to make approximations depending on the detection approach being used (such as finding patterns in the source code) and on the size and complexity of the code being analyzed. As the penalty for not detecting a vulnerability may be high, the preference is to not lose vulnerabilities, which may naturally lead to an increase in the number of

FPs [Chess and McGraw, 2004].

An obvious approach to reduce the number of FNs (*i.e.*, unreported vulnerabilities) is to combine the output of diverse SATs. Combination heuristics that suggest raising an alert when one of a set of SATs reports an issue was already addressed in the literature [Algaith et al., 2018]. However, this leads to a significant increase in the FPs as it multiplies the potential sources for vulnerability detection mistakes. Hence, alternative techniques are needed to reduce false alerts.

ML techniques have been used in several areas, including software engineering. Their application reveals good results for classification problems, and some studies have already used ML to detect vulnerabilities [Alves et al., 2016b; Russell et al., 2018; Walden et al., 2014]. However, to the best of our knowledge, no study has explored the use of SCA outputs as input for the ML classifiers.

The contribution is twofold:

- **Explore the use of ML classification algorithms to detect software vulnerabilities using the output of different SATs**, aiming at reducing the number of FPs without compromising the ability to detect vulnerabilities;
- **Validate the possibility of creating ranked lists of vulnerable source code files using the output of SATs** to guide the analysis by software development teams. This is particularly important in projects with a high number of files, as well as with time and budget constraints, which is the reality of most projects.

To support our study, we use a dataset of vulnerabilities detected by five SATs on a large set of WordPress plugins developed in PHP [Nunes et al., 2017]. We focus the study on two of the most critical vulnerability types for web application security, SQLi and XSS [van der Stock et al., 2017]. Compared with the traditional 1ooN heuristic (where an alert is raised when 1 of N detectors raises an alarm [Algaith et al., 2018]), results show an improvement for the case of SQLi vulnerabilities (namely a reduction in the false alarms). On the other hand, for XSS vulnerabilities, the results using ML are equal to the ones obtained using 1ooN. Additionally, results show that it is possible to create a ranking of the source code files considering an estimation of the potential number of vulnerabilities in each file, as reported by the multiple SATs (a clear improvement when compared to the use of the alerts raised by any SAT individually). In practice, this information can be used by the developers to prioritize their work and focus on the files with more vulnerabilities.

The rest of this appendix is structured as follows. Section A.1 presents background and related work. The exploratory study to predict vulnerabilities from the SATs output is presented in Section A.2. The approach for ranking vulnerable files is presented in Section A.3. Section A.4 discusses threats to validity, and Section A.5 concludes this study and puts forward ideas for future work.

A.1 Background and Related Work

This section presents background concepts and related work on vulnerability detection using SCA and ML algorithms.

A.1.1 Vulnerability Detection and Static Code Analysis

There are several techniques to detect software vulnerabilities. A survey by *Liu et al.* [Liu et al., 2012] details four of the most used ones: SCA, fuzzing, SPT, and VDM. SCA is the “evaluation of a system or component without the program execution” [Chess and West, 2007]; *fuzzing* is a randomized testing technique that generates random character streams for the tests; *SPT* simulates attacks of malicious users; and *VDMs* are based on software reliability models, and “specify the general form of the dependence of the vulnerability discovery process on the principal factors that affect it”. The present study focuses on static code analysis as it does not require the execution of the software to detect vulnerabilities and is typically able to achieve a high detection rate.

SATs are used to decrease the time needed for security code reviews, which may involve many people and thus be very expensive [Louridas, 2006]. Such tools are able to cover the entire source code and can thus be used early in the SDLC. The outcome of the SATs are alerts, reporting issues of diverse types, such as memory errors, resource leaks, violation of APIs or framework rules, exceptions, encapsulation violations, race conditions, and security vulnerabilities.

Different SCA techniques can be used to identify potential issues in the code [Austin et al., 2013]. The simplest one is to *scan the source code for simple patterns* [Chess and McGraw, 2004]: if a piece of code matches a rule that indicates a problem, then an alert is raised. *Data flow analysis* is a technique [Ayewah et al., 2008] where the SAT uses the possible values that variables can have to evaluate if an exception (*e.g.*, a null pointer exception) can happen at runtime. A Java example can be seen below, where the third line will raise an exception in case the variable `g` is `null`.

```

1 | if (g != null)
2 |     paintScrollBars(g, colors);
3 |     g.dispose();

```

A technique used for detecting vulnerabilities is *taint propagation*. It is a data flow technique that involves tracking tainted sources, such as user inputs, and validating if specific computations are affected by them [Chess and West, 2007]. SATs use *taint propagation* to detect SQLi and XSS vulnerabilities. When a SQLi vulnerability is exploited, a SQL statement is altered, and an attacker can read or modify the database content. An example of a SQL construct that contains a SQLi vulnerability can be seen below:

```
1 <?php
2 $email = $_GET['email'];
3 $password = $_GET['password'];
4
5 $mysqli = new mysqli('localhost', 'dbuser',
6 'dbpasswd', 'sql_injection_example');
7 $sql = "SELECT * FROM users WHERE " .
8 "username=$email AND password=$password";
9
10 if ($result = $mysqli->query($sql)) { /* code */ }
11 ?>
```

For example, if the malicious string ' OR 1=1 - is used by the attacker instead of an actual email, the expression will be evaluated as true due to the 1 = 1 comparison (tautology). As a consequence, the attacker may gain access to the system if such construct is used to support authentication. To avoid this vulnerability, the inputs should be validated, parameterized queries should be used, and the user data should be sanitized.

A XSS vulnerability allows attackers to inject malicious client-side scripts into web pages that will be viewed by other users. Such vulnerabilities are due to the lack of proper validation or escaping of the user-supplied data and are found in around two-thirds of all web applications [van der Stock et al., 2017]. When this vulnerability is successfully exploited, the attacker is able to execute scripts remotely in the user web browser.

SQLi and XSS attacks target vulnerable SSs [Kieyzun et al., 2009]. A Sensitive Sink (SS) is a command that uses external data (*e.g.*, data provided by the user), also called an entry point. For instance, a SQL query that contains entry point(s) is a SS. If one of those entry points is unsafe, it can lead to a SQLi vulnerability. The same happens for XSS vulnerabilities. For instance, the PHP command `echo "$name $city"` has two entry points that may lead to two distinct XSS vulnerabilities [Nunes et al., 2018].

Several studies compared tools to detect vulnerabilities. For example, *Antunes et al.* compared the effectiveness of four penetration testing tools with three SATs for detecting SQLi vulnerabilities [Antunes and Vieira, 2009b]. The tests were performed on top of vulnerable and non-vulnerable web services, and results show that SATs are able to achieve a higher coverage of vulnerabilities. Nevertheless, both SATs and penetration testing tools reported a considerable number of FPs. Other studies on the topic include the comparison of a SAT with a penetration testing tool [Scandariato et al., 2013], and the comparison of different vulnerability detection techniques [Austin and Williams, 2011].

Algaith et al. combined the output of different SATs to decrease the FNs [Algaith et al., 2018]. The results were obtained from the alerts raised by five SATs on a large set of WordPress plugins developed in PHP. Three combination approaches to identify SQLi and XSS vulnerabilities were used: 1) *1-out-of-N*: raises the alert when any of the tools report the alert; 2) *N-out-of-N*: raises the alert when all the tools report the alert; and 3) *simple majority*: raises the alert when the simple majority of the tools reports the alert. Results show that *N-out-of-N* has a better

specificity than *1-out-of-N* or *simple majority*, at the cost of a low recall. The heuristic with the best performance regarding the detected vulnerabilities is *1-out-of-N*, resulting in a very high recall. On the other hand, the number of FPs increases, which reduces the precision. The main limitation of this approach is the inability to explore different performance trade-offs, as basic and rigid combination rules are followed.

A.1.2 Machine Learning for Vulnerability Detection

In recent years, ML algorithms have been successfully used in a variety of complex problems, including vulnerability detection. Such algorithms can find complex patterns in the data and learn from them without relying on a predetermined model. Afterward, they can be used to make predictions on unseen data based on what was learned.

Considering the ML terminology, a Sensitive Sink (SS) represents a sample in the dataset and contains a label. In this case, two values are possible: actual vulnerability (positive) or not (negative). The output of the SATs for each SS are considered as features, and they are part of the sample. Thus, the vulnerability detection problem can be considered as a classification problem, which is concerned with separating data into distinct classes. Examples of classification algorithms include DT and SVM [Alpaydin, 2014].

As the complexity of any model depends on the number of inputs, reducing dimensionality (*e.g.*, *Feature Selection/Extraction*) may be important. Additionally, to help algorithms to cope with the infeasibility of very large datasets, *Instance Selection* techniques (*e.g.*, *sampling, boosting* [Alpaydin, 2014]) can be used. To deal with imbalanced datasets, which may compromise the performance of the algorithms on the minority classes, there are specific solutions such as *Undersampling* and *Oversampling*.

To get a realistic estimate of the performance of a model, two main sets of data are normally used: *train* and *test*. The training set is used for training the model, while the testing set is used to estimate its generalization error. This division is not trivial, as it may inadvertently influence the performance/representativeness of the model. Thus, several techniques have been proposed over the years (*e.g.*, *Partition/Leave-one-out, Bootstrap Methods* [Marques de Sá, 2001]).

Various studies have used ML to predict software vulnerabilities. An example is the study by Walden et al. [2014], where software metrics (such as the metrics CK [Chidamber and Kemerer, 1994]) and text mining were used as input for the RF ML algorithm. Datasets with software metrics and text mining data were created using three PHP applications (Drupal, Moodle, and PHPMyAdmin). The authors analyzed both recall and inspection ratios, and results show that a higher recall is obtained when using text mining data than when considering software metrics.

Alves et al. [2016b] used several ML classifiers (*Naïve Bayes, Decision Trees, Random Forest, and Logistic Regression*) to predict vulnerabilities in C/C++ projects using

datasets of software metrics. A dataset contains metrics obtained both before and after applying the patches for a number of vulnerabilities previously reported in the CVE repository. The following projects were considered: *Mozilla*, *httpd*, *glibc*, *Linux Kernel*, and *Xen Hypervisor*. In general, results showed a low precision (from 0.32% to 30.50%) and a wide range for recall (from 0.36% to 100.0%).

A.2 Exploratory Study of ML Techniques

This section presents the exploratory study on the use of ML classification algorithms to detect software vulnerabilities using the output of different SATs. The goal is to reduce the number of FPs without compromising the ability to detect vulnerabilities. We present the dataset used, introduce the ML algorithms tested, and discuss the results.

Figure A.1 depicts the overall process. As shown, the SATs are run in the same source code. Then, the alerts are combined in a dataset, where each sample refers to a SS and contains the output of each SAT (alert raised or not). The ML algorithms are run using the dataset, and a prediction (either as vulnerable or non-vulnerable/neutral) is given to each sample in the dataset. The performance of each ML model is analyzed considering precision, recall, and F-Measure metrics.

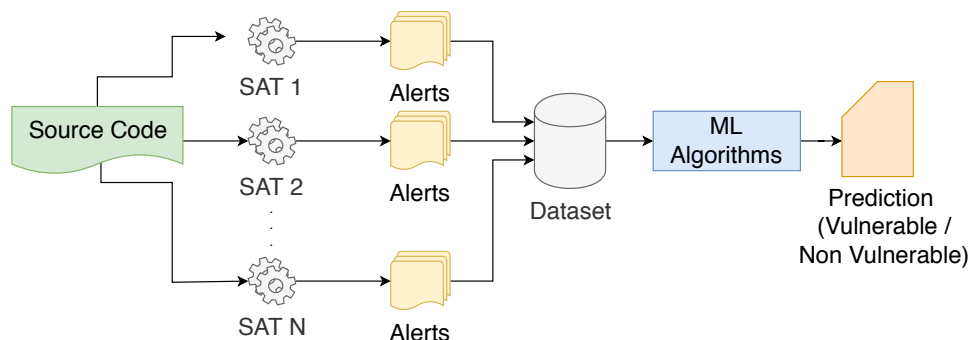





Figure A.1: Overall methodology to combine diverse SATs

A.2.1 Datasets

Two datasets were considered in this study, one with known SQLi vulnerabilities and the other with XSS vulnerabilities [WPScan, 2014]. In practice, the two datasets are based on the alerts raised by five different SATs that were used to analyze 134 WordPress plugins developed in PHP. WordPress is a widely used CMS and many websites of all sizes are built with it. *phpSAFE* [Nunes et al., 2015], *RIPS* [Dahse and Holz, 2014], *WAP* [Medeiros et al., 2014], *Pixy* [Jovanovic et al., 2006], *WeVerca* [Hauzar and Kofron, 2015]. They are well-known SATs for PHP and quite referenced in the literature as security review tools. The datasets have been created by *Nunes et al.* and first used in [Nunes et al., 2017]. In our study, we considered their updated version from [Nunes et al., 2018].

		Features				Label
		SAT 1	SAT 2	...	SAT n	
Samples	1	1	0	...	1	
	2	1	0	...	0	
	3	0	1	...	1	


	n	0	0	...	1	

Figure A.2: Representation of the dataset with samples and features

Each sample (line) in the datasets corresponds to a SQLi or XSS sensitive sink and contains the following information: *a*) file name of the SS, *b*) line number of the SS, *c*) report of the five SATs (either 1 when an alert was reported by the SAT or 0 otherwise), and *d*) label that indicates if the SS contains real a vulnerability or not. A representation of the datasets can be seen in Figure A.2. The original datasets (from Nunes *et al.* [Nunes *et al.*, 2018]) include information for all SSs in the plugins, being they vulnerable or not.

The WordPress plugins have 466,164 Logical Lines of Code (LLOC), and the number of SSs is 5,216 for SQLi, and 25,290 for XSS. From the datasets provided by Nunes *et al.*, we removed the vulnerable SSs that were not reported by any SAT, as they could lead the ML algorithms to wrong predictions. In practice, these SSs were originally identified as vulnerable not because they were reported by static analysis but after either a manual review process or through the report of the vulnerability on the WPScan Vulnerability Database (WPVD) [WPScan, 2014] (which makes them out of the scope of this study). The non-vulnerable SSs that were not reported by any tool were also removed, as they do not add any relevant information to the ML models and their elimination allows reducing the time/-model complexity. Hence, the filtered datasets contain only the samples (SSs) that have at least one alert raised by one of the five SATs. Additionally, the alerts regarding plugins that are not available in the WordPress repository anymore were also removed (20 for SQLi and 9 for XSS).

As the samples (sensitive sinks) in the datasets are already classified with two labels (vulnerable/non-vulnerable), we can classify each alert reported by each SAT according to the following: *a*) **True Negative (TN)**: a SS not reported by a SAT as a vulnerability and that is not an actual vulnerability (*not vulnerable code*); *b*) **False Positive (FP)**: a SS reported by a SAT as a vulnerability that is not an actual vulnerability (*not vulnerable code*); *c*) **False Negative (FN)**: a SS not reported by a SAT as a vulnerability that is an actual vulnerability (*vulnerable code*); and *d*) **True Positive (TP)**: a SS reported by a SAT as a vulnerability that is an actual vulnerability (*vulnerable code*).

Table A.1: Algorithms and Techniques

Parameter	Values
Feature Selection	Variance, Correlation
Sampling	Random under/oversampling, Synthetic Minority Over-Sampling Technique (SMOTE)
Algorithms	DT, NN, RF, SVM, Gradient Boosting (GB), Bagging

The datasets that resulted from the filtering discussed above contain 928 SQLi samples (670 (72.20%) vulnerable SSs and 258 (27.80%) non-vulnerable SSs), and 5,968 XSS samples (4,930 (82.61%) vulnerable SSs and 1,038 (17.39%) non-vulnerable SSs). The datasets are thus unbalanced, as there are many more samples in the positive class (vulnerable SSs) than in the negative one (non-vulnerable SSs). Hence, sampling techniques should be considered before running the ML algorithms. Normalization of the features is not needed as they are all in the same scale: they either represent the presence of an alert in a given SS (value 1) or not (value 0).

It is important to mention that, although the SATs report alerts for a particular type of vulnerability, it can not be said that each TP corresponds to a single vulnerability. In fact, an alert may correspond to more than one vulnerability if the SS uses more than one entry point (as in the SQL example in Section A.1). The results presented in this study refer to the vulnerable SSs and not the total number of vulnerabilities, as the SATs report one occurrence for each vulnerable SSs.

A.2.2 Machine Learning Algorithms and Techniques

Several ML algorithms were initially experimented using the Propheticus tool [R. Campos et al., 2019], and the ones that showed more promising results were studied in more detail considering a large list of configurations, as described in this section.

The list of algorithms used in the study, as well as the *feature selection* and *sampling* techniques applied are listed in *Table A.1* (details on each technique can be found in [Alpaydin, 2014]). Although not all the possible parameter configurations are listed in the table (due to space constraints), they can be easily obtained by combining the different values: for example, applying *variance* and *correlation* at the same time for feature selection, or *random undersampling* and *Synthetic Minority Over-Sampling Technique (SMOTE)* for sampling. The values used for the hyperparameters of each algorithm can be seen in *Table A.2*. Their values were also combined and submitted to the Propheticus tool. For example, for the DT algorithm, we have 24 configurations (as three of the hyperparameters have 2 possible values and one has 3 possible values: $2^3 * 3 = 24$).

Table A.2: Algorithms' Hyperparameters

Alg.	Hyperparameters
DT	criter.: [gini, entropy], min_samp_split: [.001, 2], max_feat.: [.1, .55, 1.0], min_samp_leaf: [.001, 1]
NN	hidden_layers: [(100,1), (100, 20), (100, 20, 10)], activation: [logistic, relu], solver: [sgd, adam], learning_rate: [constant, invscaling, adpative]
RF	estimators: [10, 50, 100, 200], max_feat.: [.1, .55, 1.0], criter.: [gini], min_samp_leaf: [.001, 1], min_samp_split: [.001, 2], bootstrap: [1, 0]
SVM	kernel: [linear, polynomial], C: [.01, .1, 1], gamma: [.1, 1], degree: [2]
GB	estimators: [50, 100, 200], learning rate: [1, .1], min_samp_leaf: [.001, 1], max_feat.: [.1, .55, 1.0], min_samp_split: [.001, 2]
Bagging	max_features: [.1, .55, 1.0], bootstrap: [1, 0], estimators: [50, 100, 200]

To evaluate the results, the datasets were divided into training and testing sets. The *training set* was used to train the model, while the *testing set* was used to evaluate it. The method to split the datasets into training and testing is Cross Validation (CV). It consists of dividing the datasets in k folds and using $k - 1$ for training and the remaining one for testing. Then, one of the folds used for training is replaced by one used for testing, and the model is trained and tested once again. After k iterations, the whole dataset has been used for training and testing [Alpaydin, 2014]. This process aims at avoiding problems like overfitting or selection bias. Note that each of the folds should be divided in a manner that the proportion of the classes remains approximately the same as in the original dataset. This process is called *stratification* [Alpaydin, 2014]. In our study, stratification aimed at maintaining not only the proportion of the classes in each of the folds (5-folds were used) but also the proportion of features (SATs in this case). For instance, if a set of Ss is reported as vulnerable by only one SATs, we make sure that these Ss are evenly distributed across the folds.

To add variability to the results, the fold partitioning was repeated 5 times per configuration tested. Each partition is called a seed, and this is used to avoid that a particular configuration presents the best results by chance. Consequently, the results reported by the Propheticus tool are the values multiplied by the seed number (5 in this case). For example, if the number of FPs is 50, the reported number is $50 * seed_number = 50 * 5 = 250$. It is important to notice that this does not affect the conclusions of the study, as the performance metrics are presented as proportions.

Precision, *Recall*, and *F-Measure* are the metrics used to characterize the performance of each model. **Precision** represents the ratio of TP among all the identified values, and it can be calculated using the equation $precision = TP / (TP + FP)$. **Recall** represents the ratio of identifying the TP among all the positive values, and it can be calculated using the equation $recall = TP / (TP + FN)$. **F-Measure** is the harmonic mean of *precision* and *recall*, and it can be calculated using the equation $F - Measure = 2 * \frac{precision * recall}{precision + recall}$.

It is important to emphasize that, when computing the performance metrics for each model, we take into account all the SSs, including the ones for which no SAT alerts were generated (and that were removed from the datasets to reduce time/model complexity, as explained in Section A.2.1). This allows comparing the results of our study with other approaches, namely with the results reported by Algaith et al. [2018] for the original datasets from Nunes et al. [2018].

A.2.3 Results and Discussion

This section presents the results obtained. We start by presenting the results using the 100N diversity heuristic. Then, we present an analysis of the dataset using Venn diagrams. Finally, we present the results of the ML algorithms and compare them with the ones obtained with the 100N heuristic.

Combination of SATs using the 100N Heuristic

Algaith et al. [2018] proposed a set of state-of-the-art heuristics to combine the output of SATs for vulnerability detection: *a) 100N*: raises an alert when any of the SATs raises the alert; *b) NooN*: raises an alert when all the SATs raises the alert; and *c) Simple Majority Voting*: raises an alert when the simple majority of the SATs raises the alert. As baseline for our exploratory study, we consider the *100N* heuristic, with the number (*N*) of SATs equal to 5. According to Algaith et al. [2018], this heuristic has the best performance when considering *recall*, as the number of FNs is the lowest one. Both *simple majority* and *NooN* have a higher precision, but that comes at the cost of a significant decrease in the number of detected vulnerabilities.

The detailed *100N* results are presented in Table A.3, and the respective performance metrics (precision, recall, and F-measure) are presented in Table A.4. Note that, the numbers slightly differ from the Algaith et al. paper [Algaith et al., 2018] as we removed some out-of-scope samples from the dataset (as discussed in Section A.2.1). As it can be noticed, the recall for both vulnerability types is high (0.931 for SQLi and 0.999 for XSS). This is related to the fact that almost all known vulnerabilities are identified by at least one of the SATs. Additionally, a good precision is obtained for both vulnerability types (0.722 for SQLi and 0.826 for XSS). As the number of FPs is higher than the number of TNs, the precision values are not as good as the recall values. This is due to the higher number of FPs raised by the use of this heuristic (that joins together the FPs of all the SATs). Although the performance metrics are already quite satisfactory, decreasing the number of the

FPs is obviously the most promising approach for improving the overall results (*Precision* and *F-Measure*).

Table A.3: Results using the 100N diversity heuristic for XSS and SQLi vulnerabilities

	TN	FP	FN	TP
SQLi	94.66% (4,570)	5.34% (258)	6.94% (50)	93.06% (670)
XSS	95.18% (20,505)	4.82% (1,038)	0.10% (5)	99.90% (4,930)

Table A.4: Metrics for the 100N diversity heuristic for XSS and SQLi vulnerabilities

	Precision	Recall	F-Measure
SQLi	0.722	0.931	0.813
XSS	0.826	0.999	0.904

Analysis of the SATs Outputs

As the metrics obtained when using the *100N* heuristic are already very good, it is important to analyze the relationship among the alerts raised by the several SATs. To this end, we generated Venn diagrams for both TP and FP, which allow understanding the number of vulnerabilities that are detected exclusively by one SAT or by multiple SATs. Figure A.3 and Figure A.4 show the Venn diagrams for SQLi and XSS vulnerabilities, respectively.

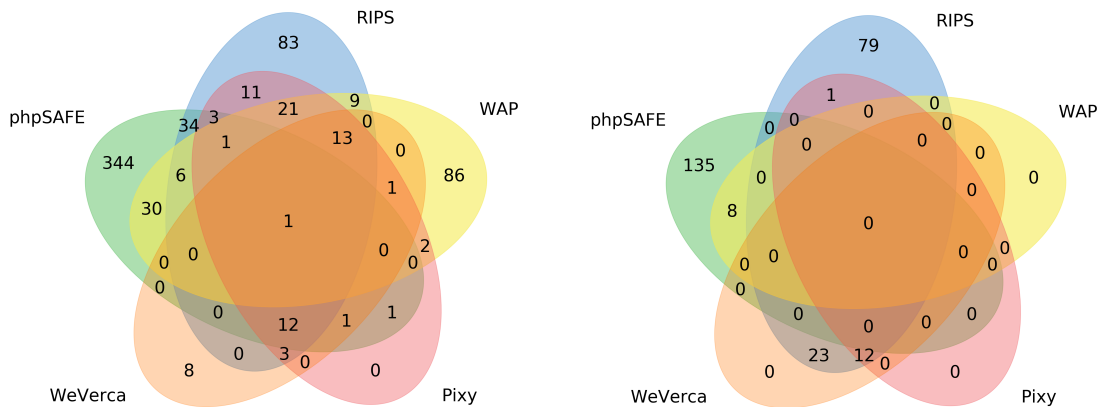


Figure A.3: Venn Diagrams of True Positives (left) and False Positives (right) for SQLi vulnerabilities

When analyzing the SQLi Venn diagrams, it can be noticed that only one true vulnerability is reported by all the SATs. Additionally, *phpSAFE* is the SAT that identified the largest number of TPs, but it is also the responsible for most of the FPs. Moreover, *Pixy* does not detect any vulnerabilities that are not reported by the remaining SATs. Overall, we can see that the different SATs complement well each other with respect to TPs. On the other hand, when it comes to FPs, a big

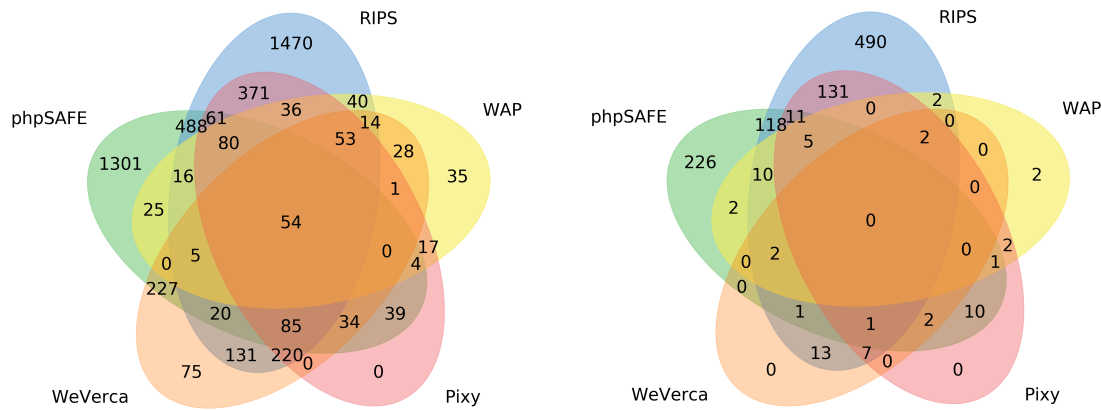


Figure A.4: Venn Diagrams of True Positives (left) and False Positives (right) for XSS vulnerabilities

overlap can be noticed. In fact, all the FPs reported by *WAP*, *Pixy*, and *WeVerca* are also reported by at least another SAT (this is why their region is equal to zero in the diagram). This means that the FPs are shared by at least two SATs. Another observation is that no TP is reported in the intersection of *RIPS-WeVerca*, but a high number of FPs are observed (23).

Regarding XSS, a higher number of vulnerabilities (54) are detected by all SATs, when compared to *SQLi*. *RIPS* is the SAT with the largest number of exclusive alerts for both TPs (1,470) and FPs (490). As noticed for *SQLi*, *Pixy* does not report any TP that is not reported by any of the remaining SATs. Additionally, neither *Pixy* nor *WeVerca* report any FP exclusively.

The likelihood of an alert being a FP is smaller when more SATs report it. The diagrams show that no FP is reported by all SATs for both *SQLi* and XSS vulnerabilities. For *SQLi*, the 4-SAT intersection also does not contain any FP, and almost all the intersection groups of three SATs do not contain FPs (except for the *RIPS-Pixy-WeVerca* intersection). Although the number of FPs for *SQLi* is small in the 4-SAT and 3-SAT intersection groups, the same conclusion cannot be made for the XSS vulnerabilities. This may be related to the size of the datasets: as there are more XSS than *SQLi* samples, the likelihood of the group intersections (alerts reported by the several tools) having items is higher (the number of groups is the same). Another possible explanation is related to the nature of the problem: SATs are usually more prepared for *SQLi* than for XSSs vulnerabilities.

Using Machine Learning

As presented before, the baseline results using the *100N* heuristic are already very good, as most of the true vulnerabilities are detected with a reasonable rate of FPs. Still, the main possibility for improvement is on the reduction of the FPs, a challenging problem to be addressed by applying ML techniques. Note that this approach also allows considering trade-offs (such as reducing the number of FPs at the cost of missing some TPs) that are not possible using the *100N* heuristic.

The results obtained are represented in the form of a Confusion Matrix (CM),

which is a graphical representation of both the actual value (rows) and the predicted value (columns). Figures A.5 and A.6 present two examples that we will discuss later in this study. As shown, the negative values (non-vulnerable - TNs and FPs) are presented in the first row, while the positive values (vulnerable - FNs and TPs) are presented in the second row. The cells contain both the percentage and the count of samples. For space reasons, in the next paragraphs we will discuss only the results of the best ML algorithm configurations for both SQLi and XSS vulnerabilities (detailed results can be found online¹).

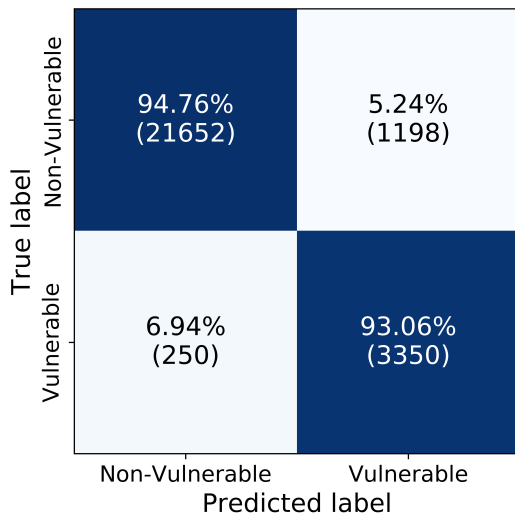
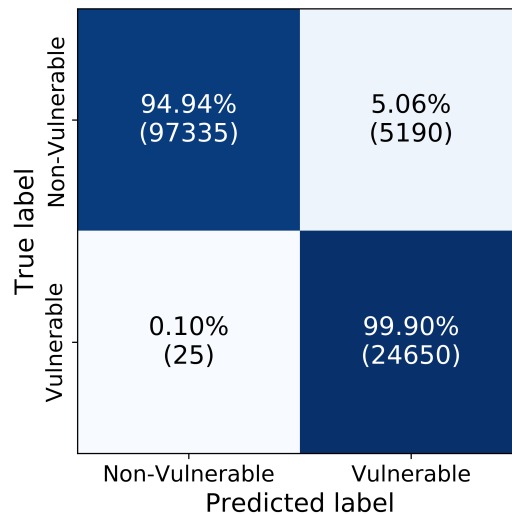
As mentioned in section A.2.2, the results are multiplied by 5 (due to the seed number used in the study). Also, even though we have removed from the dataset the non-vulnerable SSs and the vulnerable SSs without SAT alerts (to reduce complexity and because vulnerable SSs without SAT alerts are not useful in the context of this study), the results reported consider all the existing SSs. This allows creating the CMs and also comparing them with the results obtained with the *100N* heuristic.

Figure A.5 presents the CM for the best results obtained for SQLi vulnerabilities. In this case, the *Bagging* algorithm is used, with no sampling technique. The performance metrics for this case are: precision of 0.737, recall of 0.931, and F-measure of 0.822. These results are slightly better than the *100N* baseline (an increase in the precision, from 0.722 to 0.737, while maintaining exactly the same recall), which is due to the decrease of 18.4 FPs (that corresponds to 7.08% less false alarms than in the *100N* baseline - decimal values related to the fact that we average the FPs according to the number of seeds/executions). Consequently, the F-measure also increases (from 0.813 to 0.822).

Venn diagrams based on the predictions were also created to understand and interpret the improvements (not included here due to space constraints, but available online (see footnote 1). Comparing the intersection regions of the different diagrams (Figure A.3 and the Venn based on the predictions) allows understanding in which intersection the improvements are noticed. For example, we observed a reduction in the FPs, from 23 to 4.6, in the *RIPS-WeVerca* intersection/-combination. Analyzing the Venn diagram for TPs, we see that no TP is detected exclusively by *RIPS* and *WeVerca*. In general, we can conclude that the *Bagging* algorithm is able to learn this pattern, and that is why it is able to provide better results than the *100N* heuristic.

Figure A.6 presents the CM of the best configuration for XSS. This configuration uses the *RF* algorithm with no sampling technique. The performance metrics in this case are: precision of 0.826, recall of 0.999, and F-measure of 0.904. Comparing with the *100N* heuristic, we can see that the results are the same (this is also confirmed by the analysis of the different Venn diagrams). This happens because the ML algorithm classified all the SSs having any SAT alert as vulnerable. Consequently, no changes in the number of FPs or FNs are noticed, leading to the same overall performance metrics. Nevertheless, as the baseline *100N* results are already very good, we can conclude that good performance can also be achieved with ML algorithms.

¹Detailed results can be found at <https://eden.dei.uc.pt/~josep/LADC2019/>

Figure A.5: CM for SQLi vulnerabilities of the prediction (*Bagging*)Figure A.6: CM for XSS vulnerabilities of the prediction (*RF*)

In an attempt to improve the results, we removed *Pixy* from the dataset, as it does not report alerts that are not reported by the other SATs (the *Pixy* region in the Venn diagrams does not have any items/alerts, as shown in Figure A.4). However, the results obtained neither improved (confirming the vulnerable Ss or rejecting possible FPs) nor deteriorated, which means that no further conclusion can be made.

Although the results obtained with ML are not significantly better than the ones with the *100N* heuristic, for the case of SQLi vulnerabilities, there are some configurations that allow reducing the FPs at the cost of missing some TPs. This is a trade-off that cannot be explored using the *100N* approach and is very useful when the time and budget are not enough for analyzing all the potential vulnerabilities. For instance, a reduction of 24.8 FPs (9.61% reduction) can be achieved in some cases at the cost of missing 2.4 TPs (0.36% reduction). In other cases, a reduction of 35.0 FPs (13.57%) can be achieved at the cost of 3.0 TPs (0.45%). Trade-offs can also be considered for XSS vulnerabilities, although our results show a higher penalty in the TPs. For instance, a reduction of 11.2 FPs can be obtained at the cost of 31.0 TPs.

A.3 Prioritizing Vulnerable Files

As most of the projects have time and resource constraints, developers need approaches that help them prioritizing their work (in addition to having fewer FPs when analyzing SAT alerts). This way, devising a mechanism to rank the source code files in terms of the potential vulnerabilities that have to be analyzed/fixed can help development teams to focus on the ones with the biggest potential to be vulnerable. In this section, we study the possibility of creating ranked lists of vulnerable source code files using the output of SATs. The approach is based on regression algorithms, where the variables are the output of the SATs.

		Features					# Vulnerabilities
		SAT 1 #	SAT 1	...	SAT n #	SAT n	
Samples	file 1	3	1	...	2	1	3
	file 2	2	1	...	0	0	0
	file 3	7	1	...	5	1	5
	.						
	file n	1	1	...	2	1	0

Figure A.7: Representation of the file dataset with the actual vulnerability number

A.3.1 Dataset

The same set of WordPress plugins is used to create the dataset for this study. As the PHP files may have more than one vulnerability, which may result in more than one alert reported per file, we need to compute the total number of vulnerabilities per file.

The representation of the dataset can be seen in Figure A.7. The samples are the files, and the process to create the dataset consists of two steps: 1) count the number of true vulnerabilities per file for each vulnerability types; and 2) count the number of alerts reported by each SAT. In practice, the dataset includes the following features: *a*) file name; *b*) number of alerts reported by each SAT; *c*) binary value (0 or 1) that, for each SAT, indicates if the SAT reported alerts in the file; and *d*) the number of true vulnerabilities per file (equivalent to the label in the previous section). The features *b* and *c* repeat for the five SATs.

A.3.2 ML Techniques

Differently from the ML algorithms studied in the previous section, whose prediction is a categorical value in two classes (vulnerable/non-vulnerable), the prediction for the prioritization of vulnerable files requires a continuous value (number of vulnerabilities per file). The case study thus becomes a regression problem, and supervised learning algorithms should be chosen accordingly (*i.e.* one needs regression algorithms) [Alpaydin, 2014].

Three regression algorithms were studied: *linear regression* (uses the ordinary least squares method), *decision trees regression* (uses the DT classification algorithm as base for this regressor), and *Lasso* (model that estimates sparse coefficients). Due to time constraints, it was not possible to thoroughly explore and tune all the meta-learner hyperparameters. Thus, an ad-hoc approach based on the default

values of Scikit-learn is used [Pedregosa et al., 2011]. Nevertheless, the obtained results allow prioritizing the files by the potential presence of vulnerabilities per file and hence show that this is a promising approach.

The dataset is divided into training and testing sets (5-fold CV). To evaluate the results, three performance metrics were used: *a*) number of actual vulnerabilities in a Top-N set of files; *b*) proportion of vulnerabilities estimated in a Top-N set of files compared to the real number of vulnerabilities known; and *c*) overlap proportion of files in the two sets (Top-N prioritized by the regression algorithm and Top-N prioritized by the number of vulnerabilities).

A.3.3 Results and Discussion

Using the regression models, we calculated a prediction of the number of vulnerabilities for each file. The predicted value allows sorting the files by the number of predicted vulnerabilities. The results presented in this section are only for *linear regression*, as it consistently performed better than the other two approaches (*decision trees* and *Lasso*). Complete results can be found online (see footnote 1). The coefficient of determination r^2 is 0.8253 for *linear regression*.

Table A.5 shows the estimates for both SQLi and XSS using linear regression, *i.e.* actual number of vulnerabilities in the Top-50 files (sorted by the number of vulnerabilities). This corresponds to metrics *a* and *b* presented above. When analyzing the Top-50 files, we can observe that they contain a large portion of the total set of true vulnerabilities (58.06% for SQLi and 35.40% for XSS). This largely overlaps the Top-50 files when ranked by the actual number of vulnerabilities (64.03% for SQLi and 40.93% for XSS). Note that the number of files in the analysis (50 in our case) can be adjusted according to the needs of the project at hand.

Table A.5: Vulnerabilities identified by the Top-50 files using linear regression and actual number of vulnerabilities

	Vulnerability Count (%)	
	Regression	Actual
SQLi	389 (58.06%)	429 (64.03%)
XSS	1,745 (35.40%)	2,018 (40.93%)

A way to understand how the number of detected vulnerabilities increases is to analyze the progression of vulnerabilities when the number of top files (N) is enlarged. Figure A.8 shows the proportion of vulnerabilities in Top- N files as estimated by the regression algorithm in comparison to the Top- N list sorted by the number of true vulnerabilities, varying the N from 10 to 50 files, in increments of 10.

The results are quite good, as the proportion of vulnerabilities detected is always above 75.0%. For example, for the Top-10, the proportion is 76.05% for SQLi and 83.97% for XSS vulnerabilities. As it can be noticed, the proportion always increases for SQLi, but the same does not happen for XSS. For instance, the proportion decreases from the Top-10 to the Top-20 files. This is related to the fact

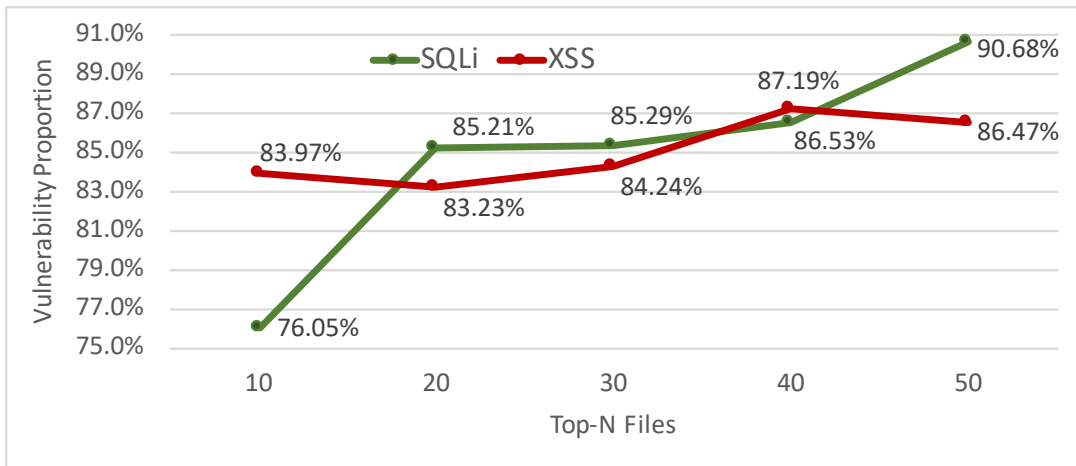


Figure A.8: Proportion of vulnerabilities detected by the Top-N files

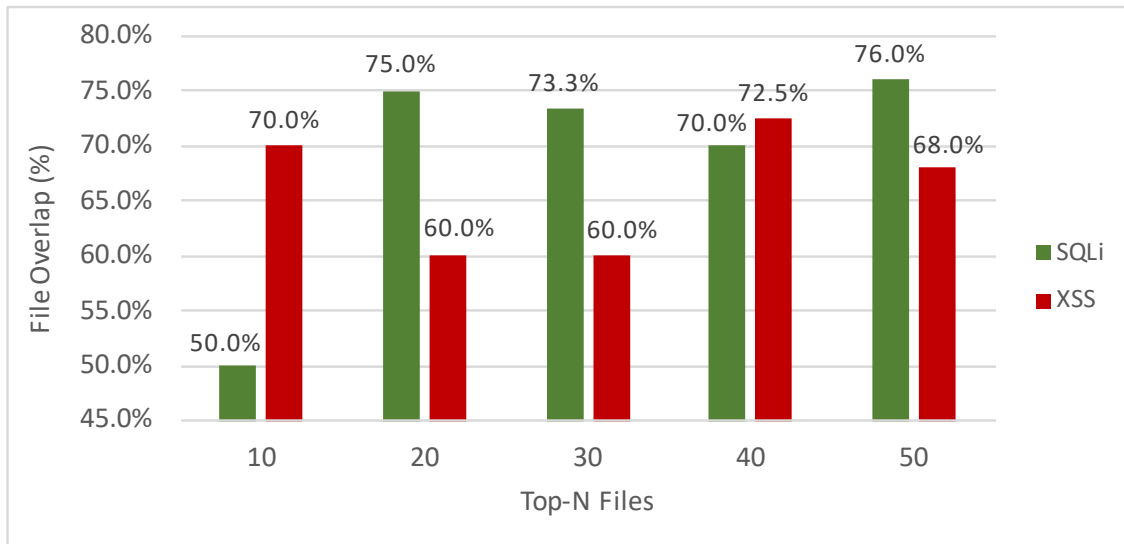


Figure A.9: Proportion of files detected compared with the Top-N files

that the selected files in that range (Top-11 to Top-20) have less predicted vulnerabilities than real values. Although the proportion decreases, the number of vulnerabilities increases as more files are added.

Another analysis that can be made is the comparison of the ordered lists of files. This can be done by analyzing the list created by the regression techniques with the list ranked by the actual number of vulnerabilities. Figure A.9 represents the overlap of files of the prioritized lists created by the regression techniques and the actual number of vulnerabilities. Although the order may not be the same, what is important is whether the two lists have a large overlap of files (metric c presented in the previous section). As can be seen in Figure A.9, the two lists have a large overlap, where the only exception is for the Top-10 files for SQLi, where the overlap is only 50.0%. For all the other cases, the overlap is larger, varying from 60.0% to 76.0%. This is an interesting outcome, as it shows that the regression-based list largely overlaps the top set of files when sorted by the number of actual vulnerabilities.

As shown, our prioritization approach provides a very good starting point to identify the files that should be analyzed first. Nevertheless, this should be combined with the output of the vulnerability detection using ML classification algorithms (see Section A.2).

A.4 Threats to Validity

The work presented here is an exploratory study on the use of ML algorithms to improve vulnerability detection. Although the algorithms used are some of the most relevant in the state-of-the-art, other algorithms may also present good results for the same problem.

The dataset was created from the SATs of various alerts. Only a small number of known vulnerabilities in the plugins were not detected by those SATs (some vulnerabilities in the original dataset from Nunes et al. [2018] were either manually detected or they were listed in the WPVD). Nonetheless, there may be unknown vulnerabilities in the source code, thus we cannot state that the dataset contains all vulnerable SSs.

The alerts were manually analyzed and classified by experts, either as vulnerable or non-vulnerable. Although they have experience in security and in SQLi and XSS vulnerabilities, they may have committed mistakes when labeling the SSs, which may influence the results of this study.

The original dataset contained vulnerabilities that are not identifiable by any of the SATs. Consequently, the ML algorithms do not have a way to predict these SSs as vulnerable. Hence, other features (either other SATs or another source of vulnerability information) can be used when running the ML algorithms.

The file prioritization does not consider how critical a vulnerability is or the likelihood of being exploited. Although the approach highlights the files that potentially contain many vulnerabilities, those may not be the ones that lead to more damage when exploited.

A.5 Conclusion and Future Work

Software vulnerabilities are a relevant problem in software as their consequences may involve data and financial losses. Thus, detecting them before releasing the software to production is of utmost importance. Approaches that combine the output of several SATs have good results, but they suffer from a key problem: a high number of FPs.

This exploratory study shows that is possible to decrease the number of FPs for SQLi vulnerabilities in a dataset that contains alerts from different SATs on WordPress plugins. Good results are obtained for XSS vulnerabilities, but they are equivalent to the *100N* heuristic. The study also shows that using linear regression based on the output of SATs allows prioritizing the files with potentially

more SQLi and XSS vulnerabilities. This can be used as a starting point of analysis by the development teams.

For future work, the same ML algorithms need to be applied considering other vulnerability detection techniques, including software metrics. Additionally, dynamic techniques can be used to either confirm an alert as a vulnerability or reject it as a FP. The prioritization mechanism also needs to be improved using alternative approaches. Finally, the same techniques need to be applied in other datasets, either of different project size, different programming languages, or of other vulnerability types. This will allow us to propose a systematic methodology in addition to confirming it for projects with different characteristics.

Appendix B

Software Metrics

Table B.1: Description of the Software Metrics (SMs) at the File level (adapted from SciTools Understand (<https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have->))

SM	Name	Description
AltAvgLineBlank	Average Number of Blank Lines (Include Inactive)	Average number of blank lines for all nested functions or methods, including inactive regions.
AltAvgLineCode	Average Number of Lines of Code (Include Inactive)	Average number of lines containing source code for all nested functions or methods, including inactive regions.
AltAvgLineComment	Average Number of Lines with Comments (Include Inactive)	Average number of lines containing comment for all nested functions or methods, including inactive regions.
AltCountLineBlank	Blank Lines of Code (Include Inactive)	Number of blank lines, including inactive regions.
AltCountLineCode	Lines of Code (Include Inactive)	Number of lines containing source code, including inactive regions.
AltCountLineComment	Lines with Comments (Include Inactive)	Number of lines containing comment, including inactive regions.
AvgCyclomatic	Average Cyclomatic Complexity	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	Average Modified Cyclomatic Complexity	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	Average Strict Cyclomatic Complexity	Average strict cyclomatic complexity for all nested functions or methods.

Continued on next page

Table B.1 – continued from previous page

SM	Name	Description
AvgEssential	Average Essential Cyclomatic Complexity	Average Essential complexity for all nested functions or methods.
AvgFanIn	Average Inputs	Average of the number of calling sub-programs plus global variables read.
AvgFanOut	Average Output	Average of the number of called sub-programs plus global variables set.
AvgLine	Average Number of Lines	Average number of lines for all nested functions or methods.
AvgLineBlank	Average Number of Blank Lines	Average number of blank for all nested functions or methods.
AvgLineCode	Average Number of Lines of Code	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	Average Number of Lines with Comments	Average number of lines containing comment for all nested functions or methods.
AvgMaxNesting	Average Nesting	Average nesting level of control constructs.
CBC (CountClassBase)	Base Classes	Number of immediate base classes. (aka IFANIN)
CBO (CountClassCoupled)	Coupling Between Objects	Number of other classes coupled to.
CountDeclClass	Classes	Number of classes.
CountDeclFunction	Function	Number of functions.
CountLine	Physical Lines	Number of all lines. (aka NL)
CountLineBlank	Blank Lines of Code	Number of blank lines. (aka BLOC)
CountLineCode	Source Lines of Code	Number of lines containing source code. (aka LOC)
CountLineCodeDecl	Declarative Lines of Code	Number of lines containing declarative source code.
CountLineCodeExe	Executable Lines of Code	Number of lines containing executable source code.
CountLineComment	Lines with Comments	Number of lines containing comment. (aka CLOC)
CountLineInactive	Inactive Lines	Number of inactive lines.
CountLinePreprocessor	Preprocessor Lines	Number of preprocessor lines.
CountPath	Paths	Number of possible paths, not counting abnormal exits or gotos. (aka NPATH)
CountSemicolon	Semicolons	Number of semicolons.
CountStmt	Statements	Number of statements.
CountStmtDecl	Declarative Statements	Number of declarative statements.
CountStmtEmpty	Empty Statements	Number of empty statements.

Continued on next page

Table B.1 – continued from previous page

SM	Name	Description
CountStmtExe	Executable Statements	Number of executable statements.
DIT (MaxInheritance-Tree)	Depth of Inheritance Tree	Maximum depth of class in inheritance tree.
FanIn	Inputs	Number of calling subprograms plus global variables read. (aka CountInput)
FanOut	Outputs	Number of called subprograms plus global variables set. (aka CountOutput)
HK	Henry Kafura Size [Henry and Kafura, 1981]	Interconnectivity of a procedure with its environment.
LCOM (PercentLackOf-Cohesion)	Lack of Cohesion in Methods	100% minus the average cohesion for package entities. (aka LCOM, LOCM)
MaxCyclomatic	Max Cyclomatic Complexity	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	Max Modified Cyclomatic Complexity	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Max Strict Cyclomatic Complexity	Maximum strict cyclomatic complexity of nested functions or methods.
MaxEssential	Max Essential Complexity	Maximum essential complexity of all nested functions or methods.
MaxMaxNesting	Max Nesting	Maximum of Maximum nesting level of control constructs.
MaxNesting	Nesting	Maximum nesting level of control constructs.
NOC (CountClass-Derived)	Number of Children	Number of immediate subclasses.
RatioCommentToCode	Comment to Code Ratio	Ratio of comment lines to code lines.
RFC (CountDeclMethodAll)	Methods	Number of methods, including inherited ones. (aka RFC: response for class)
SumCyclomatic	Sum Cyclomatic Complexity	Sum of cyclomatic complexity of all nested functions or methods. (aka WMC)
SumCyclomaticModified	Sum Modified Cyclomatic Complexity	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	Sum Strict Cyclomatic Complexity	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	Sum Essential Complexity	Sum of essential complexity of all nested functions or methods.

Continued on next page

Table B.1 – continued from previous page

SM	Name	Description
SumMaxNesting	Sum Max Nesting	Sum of maximum nesting level of control constructs.

Table B.2: Description of the Software Metrics (SMs) at the Function level (adapted from SciTools Understand (<https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have->))

SM	Name	Description
AltCountLineBlank	Blank Lines of Code (Include Inactive)	Number of blank lines, including inactive regions.
AltCountLineCode	Lines of Code (Include Inactive)	Number of lines containing source code, including inactive regions.
AltCountLineComment	Lines with Comments (Include Inactive)	Number of lines containing comment, including inactive regions.
CountInput	Inputs	Number of calling subprograms plus global variables read. Also known as FANIN.
CountLine	Physical Lines	Number of all lines. (aka NL)
CountLineBlank	Blank Lines of Code	Number of blank lines. (aka BLOC)
CountLineCode	Source Lines of Code	Number of lines containing source code. (aka LOC)
CountLineCodeDecl	Declarative Lines of Code	Number of lines containing declarative source code.
CountLineCodeExe	Executable Lines of Code	Number of lines containing executable source code.
CountLineComment	Lines with Comments	Number of lines containing comment. (aka CLOC)
CountLineInactive	Inactive Lines	Number of inactive lines.
CountLinePreprocessor	Preprocessor Lines	Number of preprocessor lines.
CountOutput	Outputs	Number of called subprograms plus global variables set. Also known as FANOUT.
CountPath	Paths	Number of possible paths, not counting abnormal exits or gotos.
CountSemicolon	Semicolons	Number of semicolons.
CountStmt	Statements	Number of statements.
CountStmtDecl	Declarative Statements	Number of declarative statements.
CountStmtEmpty	Empty Statements	Number of empty statements.
CountStmtExe	Executable Statements	Number of executable statements.

Continued on next page

Table B.2 – continued from previous page

SM	Name	Description
Cyclomatic	Cyclomatic Complexity	McCabe Cyclomatic complexity.
CyclomaticModified	Modified Cyclomatic Complexity	Modified cyclomatic complexity.
CyclomaticStrict	Strict Cyclomatic Complexity	Strict cyclomatic complexity.
Essential	Essential Complexity	Essential complexity.
Knots	Knots	Measure of overlapping jumps.
MaxEssentialKnots	Max Essential Knots	Maximum Knots after structured programming constructs have been removed.
MaxNesting	Max Nesting	Maximum nesting level of control constructs.
RatioCommentToCode	Comment to Code Ratio	Ratio of comment lines to code lines.

Table B.3: Description of the Software Metrics (SMs) at the Class level (adapted from SciTools Understand (<https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have->))

SM	Name	Description
AltAvgLineBlank	Average Number of Blank Lines (Include Inactive)	Average number of blank lines for all nested functions or methods, including inactive regions.
AltAvgLineCode	Average Number of Lines of Code (Include Inactive)	Average number of lines containing source code for all nested functions or methods, including inactive regions.
AltAvgLineComment	Average Number of Lines with Comments (Include Inactive)	Average number of lines containing comment for all nested functions or methods, including inactive regions.
AltCountLineBlank	Blank Lines of Code (Include Inactive)	Number of blank lines, including inactive regions.
AltCountLineCode	Lines of Code (Include Inactive)	Number of lines containing source code, including inactive regions.

Continued on next page

Table B.3 – continued from previous page

SM	Name	Description
AltCountLineComment	Lines with Comments (Include Inactive)	Number of lines containing comment, including inactive regions.
AvgCyclomatic	Average Cyclomatic Complexity	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	Average Modified Cyclomatic Complexity	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	Average Strict Cyclomatic Complexity	Average strict cyclomatic complexity for all nested functions or methods.
AvgEssential	Average Essential Cyclomatic Complexity	Average Essential complexity for all nested functions or methods.
AvgFanIn	Average Inputs	Average of the number of calling subprograms plus global variables read.
AvgFanOut	Average Output	Average of the number of called subprograms plus global variables set.
AvgLine	Average Number of Lines	Average number of lines for all nested functions or methods.
AvgLineBlank	Average Number of Blank Lines	Average number of blank for all nested functions or methods.
AvgLineCode	Average Number of Lines of Code	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	Average Number of Lines with Comments	Average number of lines containing comment for all nested functions or methods.
CountClassBase	Base Classes	Number of immediate base classes.
CountClassCoupled	Coupled Classes	Number of other classes coupled to. Also known as Coupling Between Objects (CBO).
CountClassDerived	Derived Classes	Number of immediate subclasses. Also known as Number of Children (NOC).
CountDeclClassMethod	Class Methods	Number of class methods.

Continued on next page

Table B.3 – continued from previous page

SM	Name	Description
CountDeclClassVariable	Class Variables	Number of class variables.
CountDeclInstanceMethod	Instance Methods	Number of instance methods.
CountDeclInstanceVariable	Instance Variables	Number of instance variables.
CountDeclInstanceVariablePrivate	Private Instance Variables	Number of private instance variables.
CountDeclInstanceVariableProtected	Protected Instance Variables	Number of protected instance variables.
CountDeclInstanceVariablePublic	Public Instance Variables	Number of public instance variables.
CountDeclMethod	Methods	Number of local methods.
CountDeclMethodAll	All Methods	Number of methods, including inherited ones. Also known as Response for a Class (RFC).
CountDeclMethodConst	Const Methods	Number of local const methods.
CountDeclMethodFriend	Friend Methods	Number of local friend methods.
CountDeclMethodPrivate	Private Methods	Number of local private methods.
CountDeclMethodProtected	Protected Methods	Number of local protected methods.
CountDeclMethodPublic	Public Methods	Number of local public methods.
CountLine	Physical Lines	Number of all lines. (aka NL)
CountLineBlank	Blank Lines of Code	Number of blank lines. (aka BLOC)
CountLineCode	Source Lines of Code	Number of lines containing source code. (aka LOC)
CountLineCodeDecl	Declarative Lines of Code	Number of lines containing declarative source code.
CountLineCodeExe	Executable Lines of Code	Number of lines containing executable source code.
CountLineComment	Lines with Comments	Number of lines containing comment. (aka CLOC)
CountLineInactive	Inactive Lines	Number of inactive lines.
CountLinePreprocessor	Preprocessor Lines	Number of preprocessor lines.
CountStmt	Statements	Number of statements.
CountStmtDecl	Declarative Statements	Number of declarative statements.

Continued on next page

Table B.3 – continued from previous page

SM	Name	Description
CountStmtEmpty	Empty Statements	Number of empty statements.
CountStmtExe	Executable Statements	Number of executable statements.
MaxCyclomatic	Max Cyclomatic Complexity	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	Max Modified Cyclomatic Complexity	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Max Strict Cyclomatic Complexity	Maximum strict cyclomatic complexity of nested functions or methods.
MaxEssential	Max Essential Complexity	Maximum essential complexity of all nested functions or methods.
MaxInheritanceTree	Max Inheritance Tree	Maximum Depth of Inheritance Tree (DIT).
MaxNesting	Nesting	Maximum nesting level of control constructs.
PercentLackOfCohesion	Percent Lack Of Cohesion	100% minus the average cohesion for package entities. Also known as Lack of Cohesion in Methods (LCOM).
RatioCommentToCode	Comment to Code Ratio	Ratio of comment lines to code lines.
SumCyclomatic	Sum Cyclomatic Complexity	Sum of cyclomatic complexity of all nested functions or methods. (aka WMC)
SumCyclomaticModified	Sum Modified Cyclomatic Complexity	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	Sum Strict Cyclomatic Complexity	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	Sum Essential Complexity	Sum of essential complexity of all nested functions or methods.