UNIVERSIDADE D
COIMBRA

Nádia Patrícia da Silva Medeiros

SOFTWARE METRICS AND MACHINE LEARNING FOR SOFTWARE SECURITY

Tese no âmbito do Programa de Doutoramento em Ciências e Tecnologias da Informação, orientada pelo Professor Doutor Marco Paulo Amorim Vieira e pelo Professor Doutor Pedro Miguel Lopes Nunes da Costa, e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Dezembro de 2022

UNIVERSIDADE D
COIMBRA

Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra

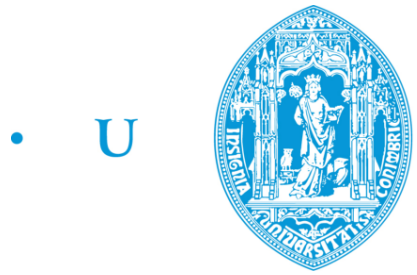
Software Metrics and Machine Learning for Software Security

Nádia Patrícia da Silva Medeiros

Doctoral Program in Information Science and Technology
PhD Thesis submitted to the University of Coimbra

Advised by Professor Marco Paulo Amorim Vieira
and Professor Pedro Miguel Lopes Nunes da Costa

December, 2022



Departamento de Engenharia Informática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Software Metrics and Machine Learning for Software Security

Nádia Patrícia da Silva Medeiros

Programa de Doutoramento em Ciências e Tecnologias da Informação
Tese de Doutoramento apresentada à Universidade de Coimbra

Orientado pelo Professor Doutor Marco Paulo Amorim Vieira
e pelo Professor Doutor Pedro Miguel Lopes Nunes da Costa

Dezembro, 2022

This work was partially supported by the European Commission (EC) through projects EUBra-BIGSEA and ATMOSPHERE, all under the Cooperation Programme, Horizon 2020, and from national funds through the Foundation for Science and Technology (FCT) in the context of the METRICS project, FCT POCI-01-0145-FEDER-032504.

- To Laura Alice with love -

Acknowledgments

THIS thesis constitutes an important milestone in my life, for which I am indebted to all the people who made it possible.

First and foremost, I would like to thank my advisors, Marco Vieira and Pedro Costa, for their invaluable advice and patience during my PhD journey. Also, for the continuous support despite the multiple tasks in which they are committed as a consequence of the important positions they hold. Specifically, I want to thank Marco Vieira for all the patience and wise advises that in different moments had great impact in my evolution as a researcher. Also, for integrating me into the Software and Systems Engineering Group of the University of Coimbra and transmitting me what was missing in my research, looking for details and teaching me how to see the opportunities of new research topics. I am also truly thankful to Pedro Costa for the initial incentive, support and motivation. He was the encourager of this great journey. I want to thank the constant guidance, availability, valuable comments and stimulating discussions over the last years.

I would like to thank Naghmeh Ivaki for her assistance at every stage of the research project. For all the availability and encouragement, the words of support and all the advises in different situations of my PhD.

I also want to offer my special thanks to all who assisted me at different stages of this research. To my colleagues and my friends.

Above all, I would like to express my gratitude to my parents and my husband. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete this journey.

Abstract

Humans face nowadays the challenge of trusting software systems for performing most of their daily tasks. The use of information technology has become so frequent that society can be seen as a combination of interactions between humans and computers with the aim of facilitating and speeding up tasks that would otherwise take a lot of time to perform. It is a well-known fact nowadays that software systems play a dominant role in diverse critical sectors, including government, financial transactions, healthcare, monitoring and control of critical infrastructures. In such sectors, a software failure (e.g., due to a successful security attack) may cause sensitive data breaches, financial losses, safety or security issues.

Modern software development projects face conflicting demands, such as, improving process quality to deliver trustworthy software products, and improving process flexibility to adapt to dynamic contexts. Thus, development teams are under an increasing pressure to deliver software products in shorter cycles with higher levels of quality. In fact, software developers often face strict deadlines due to various market pressures, and hence they usually strive to produce working software, disregarding quality aspects, including security. Designing and developing secure software is a complex endeavour and security measures should be identified and integrated from the early stages. Security by design is a concept in software engineering where software products and capabilities should be designed to be foundationally secure through several measures such as continuous testing, authentication and authorization safeguards or adherence to best programming practices.

Research regarding software security is growing, but still faces many challenges related with the difficulty on defining and computing security metrics to predict or detect unknown vulnerabilities. Furthermore, the increased importance of software security has led to the appearance of a large number of tools and techniques to detect vulnerabilities. However, despite the great advances in the standards, best practices, processes, techniques and tools used during software development lifecycle, systems are still frequently deployed with defects and vulnerabilities that may be exploited by attackers.

This thesis studies the use of **Software Metrics** and **Machine Learning** to assess the trustworthiness level of code units. The ultimate goal is to devise an approach that, based on evidence of security practices and issues in the code, **supports developers in avoiding or eliminating vulnerabilities starting from the early phases of the development process**. With such solution, it becomes possible to categorize or prioritize code units based on their trustworthiness level, warning the developers about the code units that should be reviewed before deployment, thus avoiding possible weaknesses that may be exploited by attackers. The main idea is not propose another vulnerability

detection solution, but instead focus at raising the attention of developers to the code units that are less trustworthy.

The first contribution is an *exploratory and empirical analysis on finding the best subset of software metrics to distinguish vulnerable from non-vulnerable code units*. This study includes a statistical correlation analysis of software metrics and security vulnerabilities, a dimension reduction process that contributes to select different groups of software metrics, and a feature selection analysis focusing on finding the best subset of software metrics in order to distinguish vulnerable code units from non-vulnerable ones. Then, we present a *comprehensive experiment to study how effective software metrics and machine learning can be on predicting/detecting vulnerable code units*. Several Machine Learning (ML) algorithms, namely, Random Forest, Extreme Boosting, Decision Tree and Linear and Radial Support Vector Machine, were used to extract vulnerability-related knowledge from software metrics collected from the source code of different representative software projects (Linux Kernel, Mozilla Firefox, Apache HTTPd, Xen HV and Glibc).

Grounded on the evidence that effectively predicting/detecting vulnerabilities with ML models on top of software metrics is not possible in most contexts, we propose a *Trustworthiness Benchmarking Framework* based on security evidences (e.g., software metrics, code smells) that can be used as indicators of software quality. The main goal is to compute the *trustworthiness score* (benchmarking criterion) of code units in order to be able to compare and rank them. This score is calculated based on the normalized value of the relevant features (i.e., security evidences) and their impact on the precision of ML classifier. The results show a sound ranking of the benchmarked code units.

With the goal of defining a mechanism that, based on evidences of security issues in the code, supports developers in the detection of potential issues during the software development process, we propose a *framework for code categorization*, in which the code units under evaluation are categorized/prioritized considering their trustworthiness level. Two different instantiations of these framework are presented: a **consensus-based decision-making (CBDM) approach** capable of grouping code units in several categories based on the classification result of several ML prediction models; and a solution based on **trustworthiness models to categorize and prioritize code**, not directly based the results of the prediction models, but on the clustering of the trustworthiness scores aggregated from alternative trustworthiness models. We were able to prioritize and categorize code units from a security perspective, considering different scenarios, showing that developers can use these approaches to make more efficient and effective decisions about the parts of code that might be problematic and require a deeper analysis and/or refactoring.

Keywords: Software Security, Security Vulnerabilities, Trustworthiness, Software Metrics, Machine Learning.

Resumo

Atualmente, a humanidade enfrenta o desafio de confiar em sistemas de software para realizar a maioria das suas tarefas diárias. A utilização da tecnologia de informação tornou-se tão frequente que a sociedade pode ser vista como uma combinação de interações entre humanos e computadores com o objetivo de facilitar e agilizar tarefas que de outra forma levariam muito tempo para serem executadas. É um fato bem conhecido que hoje em dia os sistemas de software desempenham um papel dominante em diversos setores críticos, incluindo o governo, transações financeiras, saúde e controle de infraestruturas críticas. Nesses setores, uma falha de software (por exemplo, devido a um ataque de segurança bem sucedido) pode causar violações de dados confidenciais, perdas financeiras ou problemas de segurança.

Os projetos de desenvolvimento de software modernos enfrentam demandas conflituantes, como melhorar a qualidade dos processos para fornecer produtos de software confiáveis e melhorar a flexibilidade dos processos para se adaptar ao contexto dinâmico. Assim, as equipas de desenvolvimento estão sob uma crescente pressão para entregar produtos de software em ciclos mais curtos com níveis mais altos de qualidade. Na verdade, os programadores geralmente enfrentam prazos rígidos devido a várias pressões de mercado e, portanto, geralmente se esforçam para produzir software que seja funcional, desconsiderando aspectos de qualidade, incluindo a segurança. Projetar e desenvolver software seguro é um processo complexo e as medidas de segurança devem ser identificadas e integradas desde os estágios iniciais. Segurança por design é um conceito em engenharia de software em que os produtos e recursos de software devem ser projetados para serem fundamentalmente seguros por meio de várias medidas, como testes contínuos, salvaguardas de autenticação e autorização ou adesão às melhores práticas de programação.

A investigação sobre segurança de software está aumentando, mas ainda enfrenta muitos desafios relacionados com a dificuldade de definir e gerar métricas de segurança para prever ou detectar vulnerabilidades desconhecidas. Além disso, a crescente importância da segurança de software levou ao surgimento de um grande número de ferramentas e técnicas para detectar vulnerabilidades. No entanto, apesar dos grandes avanços nos padrões, melhores práticas, processos, técnicas e ferramentas utilizadas durante o desenvolvimento de software, os sistemas ainda são frequentemente implantados com defeitos e vulnerabilidades que podem ser explorados por invasores.

Esta tese estuda o uso de **Métricas de Software** e **Aprendizagem Máquina** para avaliar o nível de confiabilidade de unidades de código. O objetivo final é criar uma abordagem que, com base em evidências de práticas de segurança e problemas no código, **ajude os programadores a evitar ou eliminar vulnerabilidades desde as fases iniciais do processo de desenvolvi-**

mento. Com esta solução, torna-se possível categorizar ou priorizar as unidades de código com base no nível de confiabilidade, alertando os programadores sobre as unidades de código que devem ser revistas antes da implantação, evitando assim possíveis fragilidades que possam ser exploradas por invasores. A ideia principal não é propor outra solução de detecção de vulnerabilidades, mas focar em chamar a atenção dos programadores para as unidades de código menos confiáveis.

A primeira contribuição consiste na *análise exploratória e empírica para encontrar o melhor subconjunto de métricas de software para distinguir unidades de código vulneráveis de não vulneráveis*. Este estudo inclui uma análise estatística de correlação entre métricas de software e vulnerabilidades de segurança, um processo de redução de dimensão que contribui para selecionar diferentes grupos de métricas de software, e uma análise de seleção de recursos com foco em encontrar o melhor subconjunto de métricas de software para distinguir as unidades de código vulneráveis das não vulneráveis. Em seguida, apresentamos uma *experiência abrangente para estudar a eficiência das métricas de software e da aprendizagem máquina na previsão/detecção de unidades de código vulneráveis*. Vários algoritmos de Aprendizagem Máquina (ML), nomeadamente, Random Forest, Extreme Boosting, Decision Tree e Linear and Radial Support Vector Machine, foram usados para extrair conhecimento relacionado com vulnerabilidades, a partir de métricas de software coletadas do código-fonte de diferentes projetos de software representativos (Linux Kernel, Mozilla Firefox, Apache HTTPd, Xen HV e Glibc).

Com base na evidência de que prever/detectar efetivamente vulnerabilidades com modelos de ML usando métricas de software não é possível na maioria dos contextos, propomos uma *Trustworthiness Benchmarking Framework* baseada em evidências de segurança (por exemplo, métricas de software, code smells) que podem ser usados como indicadores de qualidade de software. O objetivo principal é calcular a *pontuação de confiabilidade* (critério de benchmarking) das unidades de código para poder compará-las e classificá-las. A *pontuação de confiabilidade* atribuída a cada unidade de código é calculada usando o valor normalizado de recursos relevantes (por exemplo, evidências de segurança) e o seu impacto na precisão do classificador. Os resultados mostraram uma boa classificação das unidades de código comparadas.

Com o objetivo de definir um mecanismo que, baseado em evidências de problemas de segurança no código, apoie os programadores na detecção de possíveis problemas durante o processo de desenvolvimento de software, propomos uma *framework para categorização de código*, no qual as unidades de código sob avaliação são categorizadas/priorizadas considerando o nível de confiabilidade. Duas instanciações diferentes dessas estruturas são apresentadas: uma **abordagem de tomada de decisão baseada em consenso (CBDM)** capaz de agrupar unidades de código em várias categorias com base no resultado da classificação de vários modelos de previsão de ML; e uma solução baseada em **modelos de confiabilidade para categorizar e priorizar código**, não diretamente baseada nos resultados dos modelos de previsão, mas no agrupamento das pontuações de confiabilidade a partir de modelos alternativos de confiabilidade.

Conseguimos priorizar e categorizar as unidades de código do ponto de vista da segurança, considerando diferentes cenários, mostrando que os programadores podem usar estas abordagens para tomar decisões mais eficientes e eficazes sobre as partes do código que podem ser problemáticas e que exigem uma análise e/ou refatoração mais profunda.

Palavras-chave: Segurança de Software, vulnerabilidades de segurança, confiabilidade, métricas de software, aprendizagem máquina.

Foreword

The contributions of this thesis resulted in following publications in international conferences and journals:

- Medeiros, N. and Basso, T. (2016). Perception of trustworthiness on web services and applications based on privacy evidences. In *7th Latin-American Symposium on Dependable Computing (LADC)* (*fast abstract*).
- Medeiros, N., Ivaki, N. R., Costa, P. N. D., and Vieira, M. P. A. (2017b). Towards an approach for trustworthiness assessment of software as a service. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 220–223;
- Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2017a). Software metrics as indicators of security vulnerabilities. In *28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227. IEEE;
- Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2018a). An approach for trustworthiness benchmarking using software metrics. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 84–93. IEEE;
- Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2020). Vulnerable code detection using software metrics and machine learning. *IEEE Access*, 8:219174–219198;
- Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2021). An empirical study on software metrics and machine learning to identify untrustworthy code. In *17th European Dependable Computing Conference (EDCC)*;
- Medeiros, N., Ivaki, N. R., Costa, P. N. D., and Vieira, M. P. A. (2022). Trustworthiness models to categorize and prioritize code for security improvement. *Journal of Systems and Software (JSS), doi.org/10.1016/j.jss.2023.111621.*.

The work detailed in this thesis was accomplished at the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC), within the context of the following projects:

- **EUBra-BIGSEA** - Europe – Brazil Collaboration of BIG Data Scientific Research through Cloud-Centric Applications; a 24-month project aiming at providing services in the cloud for the processing of massive data coming from highly connected societies, which impose multiple challenges on resource provision, performance, Quality of Service and privacy. Processing those data require rapidly provisioned infrastructures customised to Big

Data requirements. EUBra-BIGSEA were funded by European Commission and Brazilian Ministry of Science, Technology, and Innovation.

- **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).

Contents

List of Figures

List of Tables

Acronyms

AIJ Aggregation of Individual Judgments

AIP Aggregation of Individual Priorities

CLARA Clustering Large Applications

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

CWE Common Weakness Enumeration

CBDM Consensus-Based Decision-Making

CI Continuous Integration

CBO Coupling Between Objects

XSS Cross-site scripting

DM Data Mining

DT Decision Tree

EM Eigenvalue Method

Xboost Extreme Gradient Boost

FN False Negative

FP False Positive

FPR False Positive Rate

HK Henry Kafura

HTTP Hypertext Transfer Protocol

HV Hypervisor

IT Information Technology

IQR Interquartile Range

IDPS Intrusion Detection and Prevention Systems

LCOM Lack of Cohesion

LR Logistic Regression

LF Lower Fence

ML Machine Learning

MDA Mean Decrease Accuracy

MDG Mean Decrease Gini

NB Naive Bayes

N Negative

NFRs Non-Functional requirements

OSS Open Source Software

OWASP Open Web Application Security Project

ODC Orthogonal Defect Classification

PAM Partitioning Around Medoids

P Positive

PbD Privacy by Design

QbD Quality by Design

RF Random Forest

RATS Rough Auditing Tool for Security

RGMM Row Geometric Mean Method

SSDLC Secure Software Development Lifecyce

SAM Security Assessment Model

SAW Simple Additive Weighting

SDLC Software Development Lifecyce

SMs Software Metrics

SQA Software Quality Assurance

SATs Static Analysis Tools

SQL Structured Query Language

SVM Support Vector Machine

TN True Negative

TP True Positive

TPR True Positive Rate

TMR Trustworthiness Model-Based Rank

TMs Trustworthiness Models

UF Upper Fence

Chapter 1

Introduction

MODERN organizations and infrastructures are backed by software systems that execute critical operations and transactions, providing services and dealing with huge amounts of sensitive data for supporting effective decisions and constant business/system adaptation. Besides that, it is a well-known fact that software systems play a dominant role in our daily life in diverse critical sectors, including government, financial transactions, healthcare, monitoring and control of critical infrastructures. In such sectors, software failures, either caused by internal bugs or by successful security attacks, can cause more damages than ever before, with serious effects in the disclosure of confidential data, safety of the users and integrity issues. Therefore, building trustworthy and secure software is critical for the success of organizations. This tremendously increased challenge is driving researchers and businesses to come up with tools, techniques, standards, and regulations to help developers to improve software security.

Security has been and will continue to be one of the key concerns in most critical software systems and applications. In fact, it has been shown by several research works that the central and critical source of security breaches are software vulnerabilities such as, Buffer overflow, Structured Query Language (SQL) injection or Cross-site scripting (XSS) (Wagner et al. [2000]; McGraw [2006]). Also, research studies show that most software vulnerabilities are either caused by the use of legacy code or by software configuration, design, and implementation mistakes made by less professional or negligent developers with a lack of knowledge about security (M.Graff and Wyk [2003]).

Software security issues should be avoided or eliminated from the early stages of the software development process, as the later the faults are discovered, the greater may be the consequences and costs of fixing them. To achieve this, developers need to be able to identify, investigate, and use the code characteristics to detect/predict security issues. In fact, integrating security within the entire development lifecycle has been proven to be the most effective way to develop

secure software (Assal and Chiasson [2018]; Vicente Mohino et al. [2019]).

The increased importance of software security has led to the appearance of a large number of tools and techniques to detect vulnerabilities (Agrawal and Khan [2009]; Zhao and Gong [2015]; Ghaffarian and Shahriari [2017]; Vieira et al. [2009]). In practice, to prevent vulnerabilities, software developers should: *i)* apply coding best practices and perform security reviews of the code (Wagner et al. [2000]); *ii)* use static code analyzers, to examine the source code for vulnerable patterns and practices (Elia et al. [2017]; Chess and McGraw [2004]); and/or *iii)* execute penetration tests, to check for exploitable vulnerabilities during execution (e.g., by emulation of security attacks) (Arkin et al. [2005]).

The problem is that, existing techniques and tools to discover vulnerabilities or bugs in software frequently provide inaccurate results, reporting nonexistent vulnerabilities or missing the existing ones (Neto and Vieira [2011a]), forcing intensive and global code review/correction time consuming tasks, with the consequent increase in costs, and therefore proving to be ineffective and of little use. It is thus of major importance to **research new approaches, techniques and tools that help developers to avoid or eliminate software vulnerabilities**, starting from the early phases of the software development process.

1.1 Problem Statement

Despite the huge advances in standards, best practices, processes, techniques and tools used during software development lifecycle, software systems are still frequently deployed with defects and vulnerabilities that may be exploited by attackers. In fact, since the beginning of the COVID-19 pandemic, FBI reported a 300% increase in cybercrimes[1]. This increase on security attacks has led to a considerable gap between the ability of attackers and the available security skills of software developers, making security a day-to-day struggle for every software development team. To improve the current situation, we need to identify, investigate, and use the early evidences of security issues in the code, in a way that supports developers in the detection of potential issues during the software development process (i.e., design and implementation) (Evans and Larochelle [2002]).

The potential costs of a vulnerability found after deployment are four to eight times more than when the same vulnerability is dealt with prior to deployment. The National Institute of Standards and Technology estimates that around $60 billion dollars a year are wasted due to faults in software (Telang and Wattal [2007]). Also, studies on analysis techniques show that using static code analysis to discover violations can reduce production costs up to potentially even 23% (Bardas [2010]).

Static code analysis can be performed manually or by static analysis tools (SATs). Manual auditing of code is time consuming and requires skilled human code auditors with sufficient and deep knowledge regarding security vulnerabil-

[1] https://www.cybintsolutions.com/cyber-security-facts-stats/

ities and security attacks to be able to effectively examine the code. In contrast, static analysis tools encapsulate security knowledge in a way that does not require highly skilled human auditors with security expertise, thus, are faster and can be frequently used to examine the code. Nevertheless, the output of these tools still requires evaluation by experts. Furthermore, the design and performance limitations of current tools lead many development teams not to use static analysis in their development process. Some of those limitations include the lack of a concise and coherent overview, missing support for multiple repository applications and multiple languages, and the lack of standardized integration mechanisms for third-party frameworks (Walker et al. [2020]).

Penetration testing is another widely used technique to help ensuring the security of software systems. Penetration testing allows discovering vulnerabilities by simulating attacks. To do this efficiently, testers rely on automated techniques that gather input vector information about the target and analyze the responses to determine whether an attack is successful. Techniques for performing these steps are often incomplete, and the black-box approach followed frequently leaves parts of the software system untested and vulnerabilities undiscovered (Halfond et al. [2011]).

Despite all the recent advances on static analysis and penetration testing, it is well known that both have great limitations, providing a low coverage and a high false positives rate. In fact, their low effectiveness in detecting vulnerabilities has been shown in several studies (Antunes and Vieira [2009]; Neto and Vieira [2011a]).

In recent years, several works studied the use of machine learning algorithms to predict/detect software vulnerabilities automatically (Alves et al. [2016a]; Ghaffarian and Shahriari [2017]; Russell et al. [2018]). Also, the use of software metrics for training the classification models to predict vulnerabilities is not a new topic (Menzies et al. [2006]; Shin [2008]). However, although we can find several works in the literature that use machine learning algorithms combined with software metrics to detect vulnerable code (Karim and et al. [2017]; Shen [2018]), results show that such approach is not really effective specially when there is a lack of resources (e.g., time, money, and human resources) to deal with the large number of false alarms produced.

Innovative approaches for improving software security need to be researched, especially from the perspective of helping development teams to focus their analysis and testing efforts on the most relevant parts of their code base (from a security perspective). Our work contributes towards that goal, but instead of focusing on approaches for detecting vulnerabilities, we focus on the identification and categorization of code units based on their potential trustworthiness.

In fact, evaluating trustworthiness is a very complex problem mainly due to the fact that it is a subjective, dynamic and context-dependent concept. Therefore, we define trustworthiness as the worthiness of a software system being trusted. Although assessing the trustworthiness of a software system is subjective, it can be transformed into an objective notion, by carefully identifying and evaluating all relevant measurable characteristics (functional and non-functional) that may

influence customer's reliance on that software system. It is worth noting that, although trustworthiness may involve several dependability attributes (e.g., security), it is even a notion more general than dependability.

1.2 Contributions

This thesis proposes an **approach based on the use of *Software Metrics (SMs)* and *Machine Learning (ML)*, not for predicting or detecting vulnerabilities, but for *assessing the trustworthiness level (or category)* of code units**. We focus on software metrics as quality evidences as these can be collected at any time during the software development process and are commonly used as indicators of software quality (Rawat et al. [2012]). However, other kinds of security evidence (e.g., code smells and lack of best practices) can also be used in most cases.

In short, the most important contributions of this thesis are:

- An **exploratory and empirical analysis on finding the best subset of software metrics for building accurate classifier models**, allowing to distinguish vulnerable code units from non-vulnerable ones. For this, we used a feature selection technique (Guyon and Elisseeff [2003]) to select a relevant subset of software metrics from a larger set collected at different code levels (functions, files, and projects). The dataset used (for this study and for the other contributions of this work) includes a long list of software metrics (static code metrics) extracted from the source code of several representative projects developed in C/C++. To build the classifier models we followed a supervised approach, which considers that the dataset is completely labeled (each code unit is labeled as vulnerable or non-vulnerable[2]). We show that it is possible to use a smaller group of software metrics to distinguish vulnerable and non-vulnerable units of code with a high level of accuracy, however, the best subset of predictive metrics may vary from one software system to another.
- A **comprehensive experiment to study how effective software metrics and machine learning algorithms are on distinguishing vulnerable from non-vulnerable units of code**. Several machine learning algorithms were used to extract vulnerability-related knowledge from multiple combinations of software metrics, collected at different architectural levels (functions and files), and considering alternative application scenarios with different concerns regarding security (highly critical, critical, low-critical and non-critical systems). In general, the results show that using Machine Learning algorithms on top of software metrics helps to identify vulnerable code units in security-critical software systems. However, they are not helpful for low-critical or non-critical systems due to the relatively high number of false alarms reported, which bring an additional

[2] We are aware that code units labeled as non-vulnerable may indeed have unknown vulnerabilities. However, because completely checking the dataset is not feasible, we assume that those units as being non-vulnerable.

development cost frequently not affordable.

- A **framework for benchmarking the trustworthiness of software** that, based on different attributes (e.g., software metrics) and their relative importance, allow determining how trustworthy a piece of code is, moving from a subjective to an objective trustworthiness (worthiness for being trusted) notion. The proposed framework is instantiated for the case of software metrics of different types (e.g., complexity, coupling, cohesion). Our results show that such benchmark enables the characterization of files and functions and their comparison, even for very large projects. In fact, by comparing our results with an expert-based assessment, we observed that the proposed benchmark quite accurate ranking of the benchmarked files and functions.
- A **framework to support developers in the identification of the code units more prone to be vulnerable from the early phases of coding process**. The idea is to group code units into several categories, representing their trustworthiness level from a security perspective. In other words, the goal is not to predict or detect vulnerable code, but instead to be able to develop categorization mechanisms based on the trustworthiness level of the code units to call the attention of developers to the most untrustworthy (potentially insecure) ones.
- Proposal of two different categorization models that instantiate the framework above, both based on machine learning algorithms and software metrics. The first implements a **Consensus-Based Decision-Making (CBDM) approach**, where the decision on the trustworthiness level of a piece of code is made by the aggregation of the classification results obtained by several ML prediction models built using software metrics. The second is based on **Trustworthiness Models (TMs) to categorize and prioritize code**, where the decision is made by clustering the trustworthiness scores of code units, which are computed using the trustworthiness benchmark mentioned before. Results show that the Consensus-Based Decision-Making approach is able to prioritize code units from a security perspective, considering scenarios with diverse security demands, and that the approach based on Trustworthiness Models can effectively cluster code units into different trustworthiness levels. Both solutions allow developers to identify code units that are more prone to be vulnerable.

1.3 Outline of the Thesis

The remainder of this thesis is organized into five chapters.

Chapter 2 presents the most relevant background and related work, discussing key concepts regarding software security, Secure Software Development Lifecycles (SSDLCs) and trust and trustworthiness in software systems.

Chapter 3 presents the exploratory and empirical analysis on finding the best subset of software metrics to distinguish vulnerable code units from non-

vulnerable ones, and the experiment to study how effective software metrics and machine learning algorithms are in discriminating vulnerable and non-vulnerable units of code in diverse application scenarios.

Chapter 4 presents the framework to benchmark the trustworthiness of software systems and a concrete instantiation for the case of software metrics of different types (e.g., complexity, coupling, cohesion). Experimental assessment and results are presented and discussed, including aspects related with validation and generalization of the benchmark.

Chapter 5 presents the code units categorization framework based on software metrics and machine learning to identify the parts of the code that seem to be more untrustworthy, in order to help developers to decide which parts of the code are more prone to be vulnerable. After presenting the framework structure, two different instantiations are proposed and their respective experimental evaluation and results are analysed.

Chapter 6 concludes the thesis, offering a synthesis of the work and possible research paths for extending it.

Appendix A presents a complete list of complexity, coupling and cohesion, and volume metrics, including the description of each metric.

Chapter 2

Background and Related Work

NOWADAYS almost every daily life activity, from entertaining games to the control systems of nuclear sites, is backed by complex software, prone to be vulnerable, thus a successful security attack has consequences more severe than ever before. Beyond the harmful and non-negligible impact on reputation, attacks frequently lead to catastrophic failures in the target systems, sensitive data breaches, financial losses, and even safety violations. This jeopardizes the trust of end-users, customers, organizations, and businesses in software systems.

Software security is clearly a crucial issue to consider during the design and implementation of any software system. A lot of efforts focused on the definition of best practices, standards, and regulations to help developers in building high quality and secure software have been proposed. However, despite the huge advances in software development processes, is still very difficult to build software without security vulnerabilities (Cusumano [2004]).

This chapter presents the background on the identified problem, mainly by introducing the necessary concepts and discussing relevant related work.

The outline of this chapter is as follows. Section 2.1 presents the main concepts related to software security. The different stages of a Secure Software Development Lifecycle (SSDLC) are described in Section 2.2. Section 2.3 presents the key concepts related to trust and trustworthiness in software systems. A notion of machine learning algorithms for software security is presented in Section 2.4, before concluding the chapter with some key remarks.

2.1 Software Security

Software security is a set of concepts, best practices, techniques and tools aiming at protecting software against the actions of malicious actors so that the

software continues to provide the expected service under such potential risks. Any compromise to integrity, authentication and availability makes a software system insecure.

Several research studies show that software defects/vulnerabilities (e.g., Buffer overflow, SQL injection) are a central and critical source of security breaches (McGraw [2006]; Wagner et al. [2000]; Avizienis et al. [2004]) in computer systems. Such vulnerabilities are mainly caused by unprofessional or negligent developers who lack security knowledge (Elia et al. [2017]).

Research on software security has a long history and still gets a lot of attention on several topics, including security protocols and patterns to build secure systems (Fernandez-Buglioni [2013]), software security testing (Wysopal and et. al [2006]), vulnerability detection (Antunes and Vieira [2010]), attack prediction (Abdlhamed et al. [2017]), intrusion detection systems (Di Pietro and Mancini [2008]), and intrusion tolerance (Deswarte et al. [1991]), just to name some.

The frequent deployment of systems with software defects/vulnerabilities that escaped the testing phases of the software development process, emphasizes the importance of focusing on security aspects from the early stages of software development. Vulnerabilities, if not uncovered and mitigated during software development, can incur huge cost in terms of time, money and reputation after deployment. It is thus of utmost importance to integrate security within the development life cycle, as part of the software development process, in the form of an ongoing process involving people and practices, to ensure application confidentiality, integrity, and availability.

Security is most effective if planned and managed throughout every stage of software development life cycle (SDLC), especially in critical applications or those that process sensitive information. To instruct software developers to incorporate security and, in general, quality into software, there are several well-established and widely known standards and best practice recommendations, such as Software Quality Assurance (SQA) (Chemuturi [2010]), Open Web Application Security Project (OWASP) secure coding practices (Turpin [2010]), ISO / IEC 27034[1] (Poulin and Guay [2008]), and Privacy by Design[2] (PbD) (Cavoukian [2009]). However, research and experience show that modern software still fails in meeting basic security requirements (Cusumano [2004]).

2.1.1 Software Vulnerabilities

Software vulnerabilities have been widely studied over the years, but they still remain a significant threat to computer security today. In computer security, the word vulnerability refers to a weakness in a system, through which, if successfully exploited, an attacker may become capable of violating one or more

[1] Provides guidance to assist organizations in integrating security into the processes used for managing their applications

[2] Framework that calls for privacy to be taken into account throughout the whole engineering process

security properties of the system, such as confidentiality, integrity, and availability, leading to undesired consequences (McGraw and Viega [2005]).

Removing vulnerabilities after deployment is potentially very costly in terms of time, money, and effort (McGraw [2004]). For this reason, prevention of vulnerabilities should be carried out in early stages of software development. However, detecting software vulnerabilities or distinguishing vulnerable from non-vulnerable code is not trivial. The low effectiveness of vulnerability detection tools, including static code analyzers and penetration testing is a clear proof of this fact (Evans and Larochelle [2002]; Neto and Vieira [2011a]). Thus, software is often deployed with bugs that can be exploited by attackers causing system outages, data breaches, or even safety issues.

The Open Web Application Security Project (OWASP) is a nonprofit foundation that works to improve the security of software. Therefore, the OWASP Top 10[3] is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications. Companies should adopt this document and start the process of ensuring that their web applications minimize these risks. The top 10 security risks presented in 2021 are: Broken Access Control, Cryptographic Failures, Injection, Insecure Design, Security Misconfiguration, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, Security Logging and Monitoring Failures and Server-Side Request Forgery.

There are other entities whose mission is to identify, define, and catalog publicly disclosed security vulnerabilities. Common Vulnerabilities and Exposures (CVE) Details[4] is a database that combines information from several sources. It enables to browse vulnerabilities by vendor, product, type, and date. Therefore, CVE is a glossary that classifies and analyzes vulnerabilities and then uses the Common Vulnerability Scoring System (CVSS) to evaluate the threat level of a vulnerability. Thus, it is possible to identify the most reported vulnerabilities over the years. As we can observe in CVEDetails[5] the most reported vulnerabilities by type are: *Execute Code*, *Denial of Service*, *Cross-Site Scripting (XSS)* and *Buffer Overflow*.

2.1.2 Preventing Software Vulnerabilities

Security is clearly a crucial issue to consider during the design of any software system. Security patterns are increasingly being used by developers who take security into consideration during the early stages of software development. Thus, developers are expected to put the necessary effort to apply adequate software security principles and rules during the development of software systems in order to minimize the probability of the existence of software vulnerabilities (that can be exploited by security attacks).

[3]https://owasp.org/www-project-top-ten/
[4]https://www.cvedetails.com/
[5]https://www.cvedetails.com/vulnerabilities-by-types.php

Security protocols and patterns to build secure systems (Fernandez-Buglioni [2013]; Ankrum and Kromholz [2005]) and guidelines that help developers to design secure code (Acar et al. [2017]) are needed. We can find a large number of efforts in the literature focused on the definition of best practices, standards, and regulations to help developers in building high quality and secure software, such as:

- ISO/IEC 27000 (Disterer [2013]) - provides an overview of information security management systems and defines related terms (i.e. a glossary that formally and explicitly defines many of the specialist terms as they are used and should be interpreted within the ISO27000 standards);
- ISO/IEC 15408 (Potii et al. [2015]) - establishes the general concepts and principles of Information Technology (IT) security evaluation and specifies the general model of evaluation;
- Software Quality Assurance (SQA) (Galin [2004]; Chemuturi [2010]; Heimann [2014]) - the ongoing process that ensures the software product meets and complies with the organization's established and standardized quality specifications;
- OWASP secure coding practices (Turpin [2010]; Marcil [2014]) - provides a secure coding checklist which has a number of prevention techniques through which damage of different types of software attacks can be minimized and mitigated;
- ISO/IEC 27034 (Poulin and Guay [2008]) - multi-part standard (six documents or parts) that provides guidance on specifying, designing, selecting and implementing information security controls through a set of processes integrated throughout an organization's Systems Development Life Cycle/s (SDLC);
- Privacy by Design (PbD) (Cavoukian [2009]) - calls for privacy to be taken into account throughout the whole engineering process.
- Sensei (De Cremer et al. [2020]) - tries to enforce secure coding guidelines in the integrated development environment. An structured description and comparison between most of these efforts can be found in Beckers et al. [2014] and Shan et al. [2019].

Despite all these efforts, developers often discard such guidelines either fully or partially. Validating the software against such guidelines is possible but also time-consuming, which limits its applicability in reality. Thus, it is still very difficult for developers to build software without vulnerabilities. This has led to many works trying to mitigate the damage that such vulnerabilities can cause at runtime, for example, via intrusion detection systems and attack tolerance techniques (Ouffoué et al. [2019]; Abdlhamed et al. [2017]; Ouffoué et al. [2016]).

2.1.3 Detecting Software Vulnerabilities

Existing approaches for the detection of vulnerabilities in the early stages of the software development are nowadays available. For example, static code analysis (Chess and McGraw [2004]) and penetration testing (Arkin et al. [2005]).

In static code analysis, the source code (or compiled code) of a software is examined statically, without executing it. Thus static analysis tools are recurrently used by developers to search for vulnerabilities in the source code of web applications. However, distinct tools provide different results depending on factors such as the complexity of the code under analysis and the application scenario (Nunes et al. [2018]). Static analysis of code can be done manually or by using static analysis tools (SATs). Manual auditing of code is time consuming and requires skilled human code auditors with sufficient and deep knowledge regarding security vulnerabilities and security attacks to be able to effectively examine the code. In contrast, static analysis tools encapsulate security knowledge in a way that does not require highly skilled human auditors with security expertise, thus, are faster and can be frequently used to examine the code. Nevertheless, the output of these tools still requires evaluation by experts.

A first step towards automated detection of security vulnerabilities using static analysis techniques was presented in Wagner et al. [2000]. The authors focus on the buffer overrun detection problem. Also, several frameworks to detect vulnerabilities have been proposed (Monga et al. [2009]). Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities is described in Pan et al. [2017] and a new Framework of Security Vulnerabilities Detection for PHP Web Application is presented in Zhao and Gong [2015]. In Chess and McGraw [2004] is presented the importance of automate source-code security with SATs. The authors argue that the longer a vulnerability lies dormant, the more expensive it can be to fix. Over the years, several approaches to detect vulnerabilities in source code using SATs were proposed (Vieira et al. [2009]; Antunes and Vieira [2015]).

Penetration Testing (Arkin et al. [2005]) is another technique to search for vulnerabilities and is used when the code can already be executed. It is the most frequently and commonly applied of all software security best practices, in part because it's an attractive late lifecycle activity. Once an application is finished, its owners subject it to penetration testing as part of the final acceptance (Arkin et al. [2005]). It works by emulation of security attacks to check for exploitable vulnerabilities.

Both static analysis tools and penetration testing tools have limitations and their low effectiveness in detecting vulnerabilities, providing a low coverage and a high false positives rate has been shown in several studies (Neto and Vieira [2011a]; Evans and Larochelle [2002]). In Antunes and Vieira [2010] is proposed an approach to benchmark the effectiveness of vulnerabilities detection tools in web services. The authors argue that in order to improve the effectiveness of detection it is important to adequate the vulnerability detection tool to a specific context of application.

2.1.4 Evidences of Software Security Vulnerabilities

Practice shows that it is difficult to detect vulnerabilities until they manifest themselves as security failures in the operational stage of the software. Therefore, it would be very useful to know the characteristics of software code that can indicate vulnerabilities that are uncovered by at least one security failure during the operational phase of the software. Such indicators, also known as software evidence, can be used to provide some suggestions to software managers and developers to take proactive action against potential vulnerabilities and improving their code.

Several approaches are available in the literature to deal with vulnerability prediction based on evidence and data collected from the source code of software systems (Li and Shao [2019]). In practice, such evidences can be gathered from different sources, including: software metrics, code smells, lack of security best practices in the code, alerts given by static code analysers, among others.

Although there are different sources to extract information from the source code of software regarding security (evidence of security issues), several studies show that software metrics are widely-used indicators of software quality (e.g., reliability and maintainability) (Coleman et al. [1994]; Rosenberg et al. [1998]). Different works show that there is some correlation between software metrics and security vulnerabilities (Moshtari et al. [2013]; Alenezi and Zarour [2018]).

Software quality metrics may help better understanding how reliable, safe, and secure a piece of code is. We can find several works related to the detection of security issues using data mining, machine learning, and statistical techniques combined with software metrics (Ghaffarian and Shahriari [2017]). In fact, using software metrics for training models to predict software vulnerabilities is not a new topic (Briand and et. al. [2000]; Menzies et al. [2006]; Russell et al. [2018]).

A software metric is a measure of software characteristics that are quantifiable or countable. Software metrics are important for many reasons, including measuring software performance considering different characteristics. For example, complexity, coupling, cohesion and volume can be measured during software development and are used to evaluate the quality of software (Fenton and Pfleeger [1997]):

- **Complexity metrics**: refer to how complicated and unwieldy it is for a developer to understand a piece of code (e.g., Cyclomatic Complexity). High code complexity brings with it a higher level of code defects, making the code costlier to maintain (Kearney et al. [1986]).
- **Coupling metrics**: refer to the level of interconnection and dependency among software entities (e.g., Coupling Between Objects). Entities are said to be highly coupled when they depend on each other to such an extent that a change in one necessitates changes in others dependent upon it. Moreover, highly coupled entities are difficult to understand in isolation and reuse because dependant entities must be included (Chowdhury and

Zulkernine [2011]).

- **Cohesion metrics**: refer to the degree that a particular entity provides a single functionality to the software system as a whole (e.g., Lack of Cohesion). Highly cohesive entities, which have only one responsibility, are more desirable than weakly cohesive entities that do many operations and therefore are likely to be less maintainable and reusable (Chowdhury and Zulkernine [2011]).
- **Volume metrics**: refer to the size of a program (e.g., Lines of Code). Programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume.

Liu and Traore [2006] studied the empirical relationship between attackability as an external software quality attribute and coupling as an internal software attribute. The results show that there is a strong correlation between attackability and coupling metrics. Complexity metrics are also related to software security (Shin and Williams [2008]), and several works investigated the correlation of software metrics and the existence of vulnerabilities (Henrique Alves [2016]). In fact, software metrics are usually used to evaluate software quality, represented by several attributes, such as, security, availability, and maintainability (Coleman et al. [1994]). Software metrics are able to discriminate vulnerable and non-vulnerable functions, but it is not possible to find strong correlations between these metrics and the number of vulnerabilities (Alves et al. [2016b]).

Complexity, coupling, cohesion and volume related structural metrics are important indicators of the quality of software architecture, and software architecture is one of the most important and early design decisions that influences the final quality of the software system. Although some of these metrics have been successfully used as indicators of software quality, there are no systematic guidelines on how to use them to predict vulnerabilities in software.

2.2 Secure Software Development Lifecycle (SSDLC)

Current software development processes face conflicting demands: improving process quality to deliver trustworthy software products, and improving process flexibility to adapt to dynamic contexts. Software developers often face strict deadlines due to various market pressures, and hence they strive to produce working software without much regard of other aspects, including security. Thus, development teams are under an increasing pressure to deliver software products in shorter cycles with higher levels of quality (Cao [2012]).

Designing and developing secure software is a complex process and security measures should be identified and integrated while designing a software product. Besides that, software security features must be integrated in every stages of software design. Integrating security within the development life cycle has been proven to be the most effective way to develop secure software (Vicente Mohino et al. [2019]). Therefore, security can be improved as well as the quality of software.

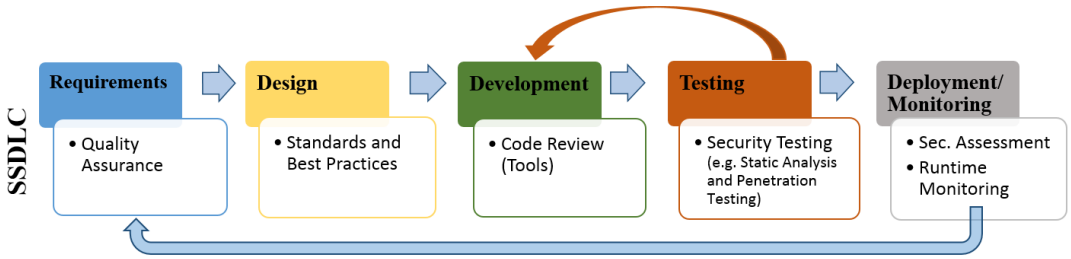


Figure 2.1: Phases of Secure Software Development Life Cycle.

Modern businesses, organizations, and critical infrastructures that want to have a Secure Software Development Life Cycle (SSDLC) should ensure that they are enabling engineering teams to get it right from the crucially important early stages of software development. However, creating a SSDLC requires dedicated effort at each phase of the software development, from requirement gathering to deployment and maintenance. In other words, SSDLCs requires that the development team focus on security at each phase of the project instead of just focusing on functionality.

Several works in the literature have been proposed in order to develop high quality and secure software using different approaches. Furthermore, it is possible to identify in which phases of software development process these approaches should be applied. We categorize and explain them based on different phases of Secure Software Development Life Cycle, such as, Requirements Definition, Design, Development, Testing and Deployment/Monitoring, as depicted in Figure 2.1, and discussed next.

2.2.1 Requirements Definition Phase of SSDLC

The **Requirements Definition Phase** of the Software Development Life Cycle (SDLC) includes the scope of work necessary to define, analyze and document business and end-user requirements. When developing under a structured type of SDLC, requirements may be further refined within Functional and Non-Functional Requirements documents. The Non-Functional requirements (NFRs) are related to the quality attributes of the software to be developed. Therefore, it is important to better understanding which software quality attributes should be considered and assured when engineering secure and trustworthy software systems (Mohammadi et al. [2014]; Hoekstra et al. [2013]).

The business and system analyses are critical and must include security parameters for programmers in software requirement specifications (Turner [2014]). Several works tried to investigate and understand how software quality can be defined and improved. Furthermore, several principles and guidelines are proposed to secure coding (M.Graff and Wyk [2003]).

In Rosenberg et al. [1998] the authors described how software metrics can be used to improve the quality and reliability of software products. In Gaffney Jr [1981] the nature of software quality, software metrics and their relationship with traditional software indicators (e.g. reliability) are presented. Another

work presented in Briand and et. al. [2000] investigate the relationship between system' design and the quality of software developed.

It becomes important to understand how software quality is influenced in order to include new specifications during the requirements phase. Also, the development teams should be aware of the guidelines to secure code in the early phases of software development, such as, how to manage Code Quality and Code Security at enterprise scale (Campbell and Papapetrou [2013]), how security can be evaluated (Criteria [1999]) (Common criteria for information technology security evaluation) or which security principles and techniques should be considered (security [2009]) (Iso 15408-1/2009, Security techniques).

Security assurances are often based on the traditional and ad hoc approach of conducting penetration tests followed by a patching process. This approach is very costly and endangers the fulfillment of the basic goals of system security, namely confidentiality, integrity, availability, and accountability. Recently, many researchers addressed security requirements engineering as an integral and essential element of systems engineering (Saleh and Elshahry [2009]).

2.2.2 Design Phase of SSDLC

In **Design Phase** the system and software design documents are prepared as per the requirement specification document in order to define the overall system architecture. Thus, the system is designed to satisfy the identified functional and non-functional requirements. The structure of the system including complete architecture diagrams along with technology details should be defined, as well as the standards and best programming practices to be applied during the development process.

2.2.3 Development and Testing Phases of SSDLC

The purpose of the **Development Phase** is to convert the system design prototyped in the Design Phase into a software system that addresses all documented system requirements. At the end of this phase, the system will enter the **Testing Phase**. Software testing is an important and critical phase of software development life cycle to find software faults or defects and then correct those faults. However, testing process is a time-consuming activity that requires good planning and a lot of resources. Therefore, techniques and methodology for predicting the testing effort is important process prior the testing process to significantly increase efficiency of time, effort and costs.

Nowadays, in order to speed up the development process and avoid building the wrong solution Development and Testing phases are made simultaneously. This is known as agile development and takes a test-first approach, rather than the test-at-the-end approach of traditional development. Since agile testing relies on regular feedback from the end user, testing and coding are done incrementally and interactively, building up each feature until it provides enough value to release to production. The main reasons to do agile testing are to save money and time.

Despite all existing efforts during **Requirements** and **Design**, software is still shipped with exploitable vulnerabilities causing huge damages to the systems and businesses.

Several bugs and vulnerabilities remain undetected for long periods when most of them are easy to avoid or detect and correct (Elia et al. [2017]). Thus, developers are often held responsible for security vulnerabilities. The real issues frequently stem from a lack of organizational or process support to handle security throughout development tasks (Assal and Chiasson [2019]). Increasingly developers are becoming aware of the importance of software security, as frequent security incidents emphasize the need for secure code.

In Agrawal and Khan [2009] is proposed a framework to identify, analyze and mitigate vulnerabilities during the development phase. This framework may be applied in conjunction with any of the software development processes in order to detect and remove vulnerabilities in each phase. In Ko et al. [1997] is presented a real-time intrusion detection system for a distributed system. This framework allows a runtime monitoring that detect exploitations of vulnerabilities in security-critical programs.

Various research efforts have targeted security testing in the past few years. The scope of concern and detection capabilities of the proposed approaches differ and can be applied during **Development** and **Testing** phases. For instance, some of the approaches involve a reasonable amount of manual intervention either by developers or by code-reviewers to reveal vulnerable code segments in the software. However, this is usually quite difficult to apply, especially for large-size software, which limits the applicability of such approaches.

2.2.4 Deployment/Monitoring Phase of SSDLC

After **Deployment** it is also important to perform continuous monitoring of software products. There are some works presented in literature propose models to evaluate and assess the security level. In Siavvas et al. [2021] is presented a hierarchical security assessment model (SAM) that allows the evaluation of the internal security of software products. The proposed approach is based on static analysis alerts and software metrics, following the guidelines of ISO/IEC 25010 (ISO, 2011). The model decomposes the notion of security into a set of security characteristics (e.g.,Confidentiality), which are further decomposed into a set of more tangible security properties (e.g., Encapsulation) that are directly quantifiable from the source code through low-level measures (i.e., static analysis alerts and software metrics). Also, the work presented in Howard et al. [2005] proposes a metric for determining whether one version of a system is more secure than another. Rather than counting bugs at the code level or counting vulnerability reports at the system level, the authors count a system's attack opportunities. They argue that in order to improve system security it is important to know the likelihood that the system will be successfully attacked to reduce its attack surface.

Intrusion Detection and Prevention Systems (IDPS) are security systems that

are used to detect and prevent security threats to computer systems and computer networks. These systems are configured to detect and respond to security threats automatically by reducing the risk to monitored computers and networks (Mudzingwa and Agrawal [2012]). As security incidents are increasing and are more aggressive, IDPS have also become increasingly necessary, they compliment the arsenal of security measures, working in conjunction with other information security tools such as malware filters and firewalls (Patel et al. [2010]).

The IDPS have mainly two methods of detection, Anomaly based and Signature based (Bashir and Chachoo [2014]). In an anomaly based technique a set of rules/activities is pre-defined for a user or a system. On the other hand, a Signature based technique has a database of already known attacks and based on this knowledge it tries to deal with the intrusions. Also, there are different types of intrusion detection and prevention systems, such as, Network-Based Intrusion Systems, Host-Based Intrusion Systems and Hybrid Intrusion Detection Systems. However, a generic technique needs to be developed that can help us to secure our software systems in any environment.

2.3 Trust and Trustworthiness in Software Systems

Trust and trustworthiness have been broadly studied in many different areas (Cho et al. [2015]), (Hardin [2002]). We can find several works in the literature focusing on trust issue in people social relationship and also regarding trust and trustworthiness in people within business environment (Slemrod and Katuscak [2002]).

In computer science, trust is a widely used term in various areas such as semantic web, game theory and agent systems (Artz and Gil [2007]). For example, considering software systems trust can be defined as a reliance of a customer on a system, that it will exhibit the expected behavior. The inherent risk is mainly based on a subjective belief, which might be formed based on past experiences with the same system. Thus, the trust level can be defined as the estimated probability of this reliance. However, the trust level is uncertain and may dynamically change. Based on the given definition, trustworthiness can be defined as the worthiness of a software system for being trusted.

Although it is differently defined in distinct areas, one of the common main goals in all of those definitions is to accurately assess the trust level as a robust basis for decision making, which turns out to be a very complex problem. In general, the complexity of the trust level assessment is primarily derived from the difficulty of evaluating trustworthiness. Thus, the first and the most important step for building trust is to establish trustworthiness (Hardin [2002]), finding a way to assess it as accurate as possible, helping to improve and, if necessary, providing a mean for comparison.

2.3.1 Trustworthy Software

Software trustworthiness is an important concern for developers, researchers, and enterprises. However, several factors make assessing trustworthiness a nontrivial task. These factors include the diversity of software systems, the large scale and high complexity of today's systems, and the subjective notion of trust and trustworthiness, because, depending on the context (e.g., critical systems or non-critical systems), different quality attributes (e.g., security or performance) may be involved in the assessment of the trustworthiness level of a system (Medeiros et al. [2018b]).

To build a trustworthy software, a set of functional and non-functional requirements should be assured (Amoroso et al. [1994]). Attributes and metrics used to evaluate the level of functionality of a software can vary from one service to another, depending on the functions of the software. However, they are independent from the environment, on which the software is running. Although the relative importance of the non-functional technical requirements depends on many aspects including criticality of the software (e.g., safety-critical or business-critical), importance of data (e.g., private data), and money involvement in the operations (e.g., back transactions), etc., they are usually related to the following mandatory quality attributes:

- **Security and Privacy**: assuring security and data privacy is of utmost importance in cloud computing. Considering the existing tendency for moving data and services to the cloud and by emerging cyber-attacks, lack of security and privacy are leading indicators of untrustworthiness.
- **Interoperability**: integrating systems and having business-to-business interactions in the cloud environment is essential, pushing the need for interoperable cloud-to-cloud services. Thus, in the cloud environment, where heterogeneity is ubiquitous and the need for having diverse systems successfully working together is unavoidable, the lack of interoperability decreases the level of trustworthiness.
- **Portability**: data portability, seen as the ability for moving data across interoperable systems, applications and cloud services, avoids vendor lock-in, which is essential in cloud environment.
- **Robustness**: with portable and interoperable cloud services, it is more likely to receive more malicious data. Thus, the robustness of services against these malicious data is essential.
- **Scalability**: many applications in cloud environment require a relatively large amount of processing and storage. Also, as the data size growth rate in most systems is quite high, the lack of scalability is crucial in cloud.
- **Performance**: in cloud computing, we are usually dealing with big data and big computing. Therefore, keeping performance at an acceptable level is also a big concern.

Therefore, the complex nature of trust in computer systems triggered many

research work, as discussed next.

2.3.2 Trustworthiness Assessment

Several benchmarks have been proposed in the past targeting different application domains with the goal of comparing systems considering specific characteristics. However, traditional and well-established measures disregard fundamental trust related aspects that are required by contemporary societies and modern computer systems. A trust based measure allows comparing the level of justifiable trust that one can put in different systems as not being susceptible to particular threats (Neto and Vieira [2011b]). For example, from a security perspective, the collection of evidence starts from a particular set of threats and accumulates information regarding how protected the system is against the accomplishment of those threats. The advantage is that understanding the threats faced by the system does not require knowing all the possible attacks that may realize them. Furthermore, the level of trust is not specific for a particular threat but instead represents the confidence level that a system as a whole conveys. In practice, the level of trust that one can put in a system is related to how the system performs the required tasks and how we perceive its execution.

Trust and trustworthiness assessment of software systems, as in other areas, is anything but simple, mainly due to the complex and dynamic nature of the cloud, variety of services (e.g., safety-critical or business-critical), large number of relevant quality attributes (e.g., security and performance), and last, but foremost, due to the subjective notion of trust and trustworthiness.

To increase users' trust in the systems they use, there is a need to develop trustworthy systems. These systems must meet the needs of the system's stakeholders with respect to security, privacy, reliability, and business integrity .

An approach for trustworthiness benchmarking is presented in Wang et al. [2006]. Here the authors describe a conceptual model for the trustworthiness of Internet-based software and Ding et al. [2012] presents a novel evidential reasoning based method for software trustworthiness evaluation under the uncertain and unreliable environment. Furthermore in Medeiros et al. [2017c] is proposed an approach for trustworthiness assessment of software as a service.

More trustworthy software systems and trust evaluation mechanisms can be found in Hasselbring and Reussner [2006] Chiregi and Navimipour [2017] as well as different trustworthiness assessment models and tools (Li [2017] Limam and Boutaba [2010]). Besides that, there is also presented in the literature a design for trustworthy software, including tools, techniques, and methodology of developing robust software (Jayaswal and Patton [2006]). Another works define metrics and measurement of trustworthy systems (Cho et al. [2016]) and indicators for measuring and improving software trustworthiness (Yang et al. [2009]).

A survey is conducted in Del Bianco et al. [2011] to understand the factors that influence trust in open source software (OSS) by users and developers. A total of 151 OSS stakeholders with different roles and responsibilities participated in

the survey. The survey results show that functionality and reliability are the most critical factors.

In a different kind of study (Alarcon et al. [2017a]), transparency and reputation were studied as factors that may influence trust perceptions as well as time spent reviewing code by professional software developers. In a previous study, the same authors (Alarcon et al. [2017b]) explored how developers assess code trustworthiness when asked to reuse an existing code base. They used an expert-based analysis to explore experienced programmers' perspectives on code reuse, and concluded that implementing software when considering factors like reputation, transparency, and performance can influence the trust in reusing existing code. In their second study (Alarcon et al. [2017a]), their findings suggest that the influence of transparency on trust perceptions are not as strong and straightforward as previously thought.

Despite the merit of these works, they are mainly focused on expert-based analysis to assess the trustworthiness of the code. Although this helps to understand the essential factors influencing trustworthiness, it cannot be applied in an automatic way and on a bigger scale. The other shortcoming of these works compared to what we are presenting in this paper is that they do not consider security as one of the main factors of software trustworthiness.

In addition to the above works, trust and trustworthiness assessment are vastly explored in the context of complex and dynamic environments, such as Cloud (Medeiros et al. [2017b]; Horvath and Agrawal [2015]; Lee and Brink [2020]). However, most of the works in this context are customer-centric and do not address the improvement of the software under development, especially from a security perspective.

2.4 Machine Learning for Software Security

Machine learning is a method of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention. Machine Learning are used in several application areas (e.g., financial services, transportation, health care) and are based on a diverse set of techniques. Therefore, through the use of statistical methods, the ML algorithms are trained to make classifications or predictions, and to uncover key insights in data mining projects. These insights subsequently drive decision making within applications and businesses.

There are several commonly used machine learning algorithms that can be applied to almost any data problem:

- **Decision Tree (DT)**: commonly and most used supervised learning technique to support decision making. Given a dataset composed of several features and target classes, by using the Decision Tree technique, a sequence of classification rules are generated to make decisions in diverse cases. To generate these rules, it uses a tree-like model to break up a

complex decision into several simpler decisions (S. and et. al. [1991]);

- **Random forest (RF)**: is one of the most popular ensemble learning algorithm. This algorithm consists of a combination of several DT-based classifiers, each one fitted on a random sample of a dataset, making it more accurate and robust to outliers and noise than a single DT-based classifier (Breiman [2001]);
- **Extreme Gradient Boost (Xboost)**: a specific implementation of the Gradient Boosting method that uses more accurate approximations to find the best tree models. Its main difference compared with random forest is that it builds one tree at a time. Each new tree helps to correct errors made by the previously trained tree. Xboost models are becoming popular due to their effectiveness at classifying complex data (Chen and Guestrin [2016]; Schapire [2002]);
- **Linear Support Vector Machine (SVM)**: SVM is another widely used supervised machine learning algorithm, which is usually used for solving classification problems with two classes. Linear SVM performs classifications by finding a line that best differentiates the target classes by maximizing the margin between them (Awad and Khanna [2015]);
- **Radial Support Vector Machine (SVM)**: a nonlinear or radial SVM applies the kernel trick to find a hyperplane (decision surface), instead of a line, to best separate two classes, when there are non-linear interactions in the data. It does a non-linear transformation on the features and converts them to a higher dimensional space to add non-linearities to the learning process (Boser et al. [1992]);
- **Logistic Regression (LR)**: it is used to estimate discrete values based on a given set of the independent variable(s). In simple words, it predicts the probability of occurrence of an event by fitting data to a logit function. Since it predicts the probability, its output values lie between 0 and 1 (Feng et al. [2014]);
- **Naive Bayes (NB)**: assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature (Rish et al. [2001]);
- **K-Means**: it is a type of unsupervised algorithm which solves the clustering problem. Its procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters). Data points inside a cluster are homogeneous and heterogeneous to peer groups (Sinaga and Yang [2020]).

Implementing Machine Learning and Data Mining techniques will allow computer to learn and be able to predict vulnerabilities (Ghaffarian and Shahriari [2017]). Using software metrics for training models to predict vulnerabilities is not a new topic (Briand and et. al. [2000]; Menzies et al. [2006]; Li et al. [2018]; Russell et al. [2018]). A survey of various machine learning algorithms with software metrics for prediction of software faults is presented in Karim and et al. [2017]. This work contributes to a consensus on what constitute effective soft-

ware metrics and machine learning method in software fault prediction. Other works tried to measuring, analyze and predict security vulnerabilities in software systems and also predict vulnerable software components (Alhazmi et al. [2007]; Neuhaus et al. [2007]). A survey of feature selection for vulnerability prediction using feature-based machine learning is presented in Li and Shao [2019].

However, despite the existence of several works related to the detection of security issues using machine learning and statistical techniques combined with software metrics (Ghaffarian and Shahriari [2017]) most of the studies have limitations, such as: works are done over a limited number of software metrics (e.g., complexity metrics) (Chowdhury and Zulkernine [2011]; Shin and Williams [2011]; Moshtari et al. [2013]), or focus on a single security issue such as buffer overflow (Ren and et al. [2019]), or are limited to a specific code unit (e.g., file/class or function/method) (Shin et al. [2010]) and a specific software project (Shin and Williams [2008]).

2.5 Summary

This chapter presented the background and related work, addressing aspect such as software security, secure software development lifecycle, trust and trustworthiness in software systems and finaly machine learning for software security.

The exploitation of software security vulnerabilities can have severe consequences. Thus, it is crucial to devise new processes, techniques, and tools to support teams in the development of secure code from the early stages of the software development process, while potentially reducing costs and shortening the time to market.

Despite all those efforts using standards, best practices, tolls and techniques to develop software, is still very difficult to build secure software. New approaches to help developers are still needed. It become crucial to adopt measures to secure software in early stages of software development. Thus, over the next chapters we propose approaches that, based on evidence of security practices and issues in the code, supports developers in avoiding or eliminating vulnerabilities starting from the early phases of the development process.

Chapter 3

Vulnerable Code Detection using Software Metrics and Machine Learning: Experimental Studies

In this chapter, we present an exploratory and empirical analysis on finding the best subset of software metrics to distinguish vulnerable code units from non-vulnerable ones, and a comprehensive experiment to study how effective software metrics are in classifying vulnerable and non-vulnerable units of code in diverse application scenarios.

To support the studies, we used the dataset from Alves et al (Henrique Alves [2016], Alves et al. [2016b]), which contains detailed information about the architecture, composing files, classes, and functions of five projects implemented in C/C++. This is also the dataset that we used in the experimental evaluations presented in the following chapters. It is important to mention that the dataset has been updated during the course of this work, so there will be updates throughout the chapters, considering the characteristics of the data at each point in time.

To study the possibility of finding the best subset of software metrics to distinguish vulnerable code units from the non-vulnerable ones, we conducted an **analysis of the correlation between software metrics and security vulnerabilities**. This includes: *(i) a statistical correlation analysis* using project-level metrics and security vulnerabilities; *(ii) a dimension reduction* that contributes to select different groups of software metrics; and *(iii) a feature selection analysis using a Genetic Algorithm*, focusing on finding the best subset of software metrics for building accurate classifier models to predict software vulnerabilities.

To understand if software metrics are discriminative enough for classifying vulnerable code units (in diverse development contexts), we present a comprehensive experiment on the use of **software metrics as features for creating**

machine learning models to detect vulnerabilities. Towards this, we *(i)* discuss the *class distribution* in the dataset; *(ii)* introduce the set of *machine learning algorithms* used; *(iii)* define the *application scenarios and decision criteria*; and finally, *(iv)* discuss the *classification results*. The characteristics of the code units misclassified by all machine learning algorithms are analyzed in detail to better understand the results of the prediction models.

The rest of this chapter is organized as follows. First, in Section 3.1 we introduce the dataset used. Then, in Section 3.2 we present the analysis of the correlation between software metrics and security vulnerabilities, which is followed by the experiment to study how effective software metrics and machine learning algorithms are to detect vulnerabilities in Section 3.3. The chapter closes with a summary of the main observations and results in Section 3.4.

3.1 Dataset Characteristics

The dataset from Alves et al (Henrique Alves [2016]) is the most complete one available that fits our purpose. The data, including a long list of software metrics, was extracted using the Understand tool (SciTools [2017]), from the source code of five software projects implemented in C/C++: Mozilla Firefox (mozilla.org), Apache httpd (httpd.apache.org), Linux Kernel (kernel.org), Xen Hypervisor (xen.org), and Glibc (gnu.org/software/libc). These are important and representative projects, from a security point of view: they have been used by many worldwide users and they were targeted by many security attacks. Moreover, each project is representative of a broader class of software in a particular category, in terms of functionality (e.g., the Apache httpd represents HTTP servers, like Oracle and IBM HTTP servers).

The dataset comprises a large number of software metrics of different types, including complexity (e.g., Cyclomatic Complexity), volume (e.g., Lines of Code), coupling (e.g., Coupling Between Objects), and cohesion (e.g., Lack of Cohesion) metrics. In our study, we use a total of **28 function-level** metrics, **51 file-level** metrics, and **54 project-level** metrics. Among the project level metrics, 51 are the average values of the corresponding file-level metrics, and the remaining three correspond to the total lines of code in the project (*CountTotalLOC*), total number of functions(*CountTotalFunctions*), and total number of files (*CountTotalFiles*). The dataset contains information regarding each of these metrics for each new version of the aforementioned projects from the year 2000 to 2016.

Our work focuses different architectural levels of the projects, including the entire **project**, **files**, and **functions**, each one having its own set of metrics. It is worth mentioning that class-related metrics are not considered in our work as only one of the projects (Mozilla Firefox) is implemented in an object-oriented language, C++ in the case, and therefore contains classes and methods (which were excluded from the dataset for our work). The complete list of the software metrics used, including a short description of each, is presented in tables 3.1 and 3.2 for files and functions, respectively. The complete list of software metrics

Table 3.1: File-level metrics.

Software Metrics	Description
SumEssential	Sum of essential complexity of all nested functions
MaxEssential	Maximum essential complexity of all nested functions
SumCyclomaticStrict	Sum of strict cyclomatic complexity of all nested functions
CountStmtExe	Number of executable statements
SumCyclomatic	Sum of cyclomatic complexity of all nested functions
CountLineCodeExe	Number of lines containing executable source code
AltCountLineComment	Number of lines containing comment, including inactive regions
CountLineCode	Number of lines containing source code
AltCountLineBlank	Number of blank lines, including inactive regions
CountLineBlank	Number of blank lines
AvgEssential	Average Essential complexity for all nested functions
CountLine	Number of all lines
MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions
CountStmt	Number of statements
CountLinePreprocessor	Number of preprocessor lines
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions
AltCountLineCode	Number of lines containing source code, including inactive regions
SumCyclomaticModified	Sum of modified cyclomatic complexity of all nested functions
CountDeclFunction	Number of functions
CountLineInactive	Number of inactive lines
CountSemicolon	Number of semicolons
CountLineComment	Number of lines containing comment
MaxCyclomatic	Maximum cyclomatic complexity of all nested functions
CountLineCodeDecl	Number of lines containing declarative source code
RatioCommentToCode	Ratio of comment lines to code lines
CountStmtDecl	Number of declarative statements
AvgLine	Average number of lines for all nested functions
CountStmtEmpty	Number of empty statements
AvgCyclomaticStrict	Average strict cyclomatic complexity for all nested functions
AvgLineCode	Average number of lines containing source code for all nested functions
MaxFanIn	Maximum number of calling subprograms plus global variables read
AltAvgLineCode	Average number of lines containing source code for all nested functions, including inactive regions
AvgFanIn	Average number of calling subprograms plus global variables read
MaxFanOut	Maximum number of called subprograms plus global variables set
CountPath	Number of possible paths, not counting abnormal exits
AltAvgLineComment	Average number of lines containing comment for all nested functions, including inactive regions
AvgLineBlank	Average number of blank for all nested functions
AvgLineComment	Average number of lines containing comment for all nested functions
HK	HK measures information flow relative to function size
AltAvgLineBlank	Average number of blank lines for all nested functions, including inactive regions
MaxNesting	Nesting level of control constructs
FanIn	Number of calling subprograms plus global variables read
FanOut	Number of called subprograms plus global variables set
AvgFanOut	Average number of called subprograms plus global variables set
AvgMaxNesting	Average of maximum nesting level of control constructs
SumMaxNesting	Sum of maximum nesting level of control constructs
AvgCyclomaticModified	Average modified cyclomatic complexity for all nested functions
AvgCyclomatic	Average cyclomatic complexity for all nested functions
MaxMaxNesting	Maximum nesting level of control constructs
CBO	Coupling Between Objects
LCOM	Lack of Cohesion in Methods (100% minus the average cohesion)

Table 3.2: Function-level metrics.

Software Metrics	Description
CountOutput	Number of called subprograms plus global variables set
CountLineCodeDecl	Number of lines containing declarative source code
MaxNesting	Maximum nesting level of control constructs
CountInput	Number of calling subprograms plus global variables read
AltCountLineBlank	Number of blank lines, including inactive regions
Knots	Measure of overlapping jumps
CountLineBlank	Number of blank lines
CountLineCode	Number of lines containing source code
MinEssentialKnots	Minimum Knots after structured programming constructs have been removed
AltCountLineComment	Number of lines containing comment, including inactive regions
MaxEssentialKnots	Maximum Knots after structured programming constructs have been removed
CyclomaticStrict	Strict cyclomatic complexity
CountSemicolon	Number of semicolons
CountLineComment	Number of lines containing comment
CountStmtDecl	Number of declarative statements
Cyclomatic	Cyclomatic complexity
CountLine	Number of all lines
CountLineCodeExe	Number of lines containing executable source code
CyclomaticModified	Modified cyclomatic complexity
RatioCommentToCode	Ratio of comment lines to code lines
CountPath	Number of possible paths, not counting abnormal exits
AltCountLineCode	Number of lines containing source code, including inactive regions
CountStmtExe	Number of executable statements
CountStmt	Number of statements
Essential	Essential complexity
CountLinePreprocessor	Number of preprocessor lines
CountLineInactive	Number of inactive lines
CountStmtEmpty	Number of empty statements

considering different types (complexity, volume, coupling and cohesion) and their description can be found in Appendix A.

The dataset also includes detailed information about the known vulnerabilities in the projects (disclosed between 2000 and 2016), obtained by analyzing of a large number of security patches gathered from various sources (i.e., CVEDetails, Mozilla Foundation Security Advisores (MFSA), and Xen Security Advisores (XSA)) (Alves et al. [2016b]). It is important to mention that the source of information regarding the vulnerabilities in the projects is limited to security reports. Consequently, the functions and files without reported vulnerabilities and that are labeled in the dataset as non-vulnerable are not necessarily flawless (with this in mind, we still refer to them as non-vulnerable).

Table 3.3 presents a summary of the projects and their reported vulnerabilities. It is worth mentioning that only source files (i.e., .c and .cpp files) are considered in our analysis, so the number of functions, files and lines of code presented do not include the information of C header files (i.e., .h files). Besides that each project contained only one record for each file and each function. More details about the dataset can be found online (Henrique Alves [2016]).

All the types of vulnerabilities in the dataset and their distribution across the five

Table 3.3: Summary of the dataset used.

Project	Language	Number of Files			Number of Functions			Number of LOC	# reported vulnerabilities
		Total	Vulnerable	% Vuln.	Total	Vulnerable	% Vuln.		
Mozilla Firefox	C++	27468	830	3,02%	612327	2107	0,34%	8279044	5129
Linux Kernel	C	36112	842	2,33%	830571	1229	0,15%	15422831	1229
Apache httpd	C	810	29	3,58%	9806	33	0,34%	302784	91
Xen HV	C	1257	71	5,65%	20607	154	0,75%	437150	205
Glibc	C	9785	31	0,32%	13302	16	0,12%	561411	69

projects analyzed are presented in Figure 3.1. In general, `Denial of Service` and `HTTP Response Splitting` are, respectively, the most (31.4%) and the least (0.01%) frequent vulnerabilities, although different types of vulnerabilities are scattered differently depending on the project:

- `Denial of Service` is the most popular vulnerability in the Linux Kernel;
- In Mozilla Firefox, `Execute Code` is the most frequent one;
- `Memory Corruption` was never reported in Apache httpd and Glibc, but is quite frequent in Mozilla Firefox.

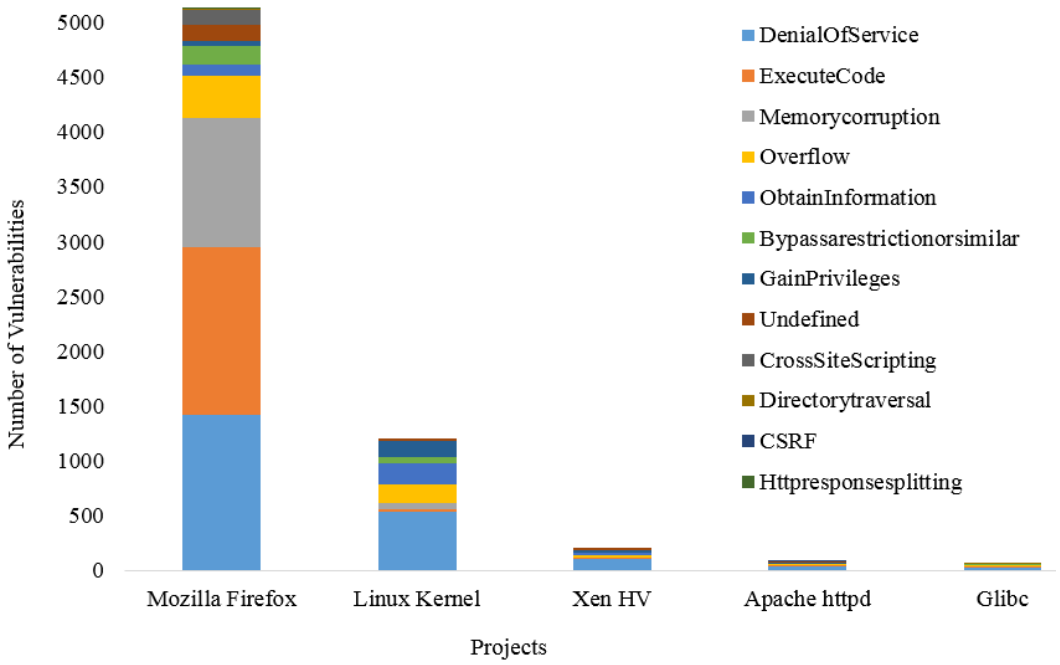


Figure 3.1: Distribution of vulnerabilities in different projects.

Despite the many advantages, there are several drawbacks associated with the use of this dataset, that should be pointed out. First, the source of information regarding the vulnerabilities of the projects is limited to the security reports. Consequently, the functions and files without reported vulnerabilities are not necessarily flawless. Second, all the projects in the dataset are implemented in C/C++, while each programming language has its own characteristics and

may differently influence software security (Turner [2014]). Consequently, our analysis may not be representative for software implemented in other languages (e.g., Java).

The dataset is quite imbalanced. The number of vulnerable code untis is very low in all software projects. However, as mentioned before, the dataset has been updated over time (either because there were changes in the dataset or refinements in the data search). These changes will be shown throughout the next chapters whenever necessary, for a better understanding of the experiments. The data presented in Table 3.3 reflects the dataset used to support the experimental study that is described in the next section.

3.2 Correlation between Software Metrics and Security Vulnerabilities

To be able to use software metrics for detecting or indicating vulnerable code, it is important to find out which metrics are somehow correlated to the quality of the code, from a security perspective. Despite some software metrics may contain useful information to distinguish vulnerable from non-vulnerable code units, others might be irrelevant or redundant. Therefore, three sets of experiments are presented, as discussed next.

We start with a **statistical analysis** aiming at finding the relationship between the internal characteristics of the software projects, represented by *project-level metrics*, and the number of vulnerabilities known. For this purpose, both Pearson (Benesty et al. [2009]) and Spearman (Myers and Sirois [2006]) correlation coefficients are used. While the first (Pearson) evaluates the linear relationship between project-level metrics and the number of vulnerabilities, the second (Spearman) evaluates the monotonic relationship between them.

In a second step, we discuss the **dimension reduction** problem. In order to build a high performance classification model out of software metrics for vulnerable code detection, it is important to search for the most informative and discriminative metrics and to discard the redundant or irrelevant ones, which may reduce the accuracy and the computational efficiency of the classifier (Liu et al. [2005]). Again using both Pearson (Benesty et al. [2009]) and Spearman (Myers and Sirois [2006]) correlation coefficients, this process allows us to reduce the number of features under consideration (in this case, the number of software metrics).

A **feature selection analysis**, based on a Genetic Algorithm, is presented to understand the best subsets of software metrics considering both *file-level* and *function-level metrics*, without loss of useful information, by eliminating irrelevant and redundant metrics with little or no predictive information. The resulting subsets of metrics may allow improving the accuracy and comprehensibility of vulnerability detection/prediction tools.

The three experiments were carried out independently (i.e., they do not depend on each other) and the results and conclusions obtained will be used as the basis

for the following chapters (e.g., the best subset of software metrics obtained by the feature selection analysis is used as input for the trustworthiness benchmark presented in Chapter 4).

The R Project (Team [2017]) was used to run the experiments: R Caret (Kuhn et al. [2020]) and RandomForest (Kuhn [2016]) packages, respectively providing the genetic algorithm and the random forest classification algorithm to implement our methodology. All the experiments were executed on virtual machines with Ubuntu 16.04, a 2.0 GHz Intel Xeon E312xx (Sandy Bridge) processor, 4GB RAM and 16MB cache.

3.2.1 Statistical Analysis

To calculate the correlation between project-level metrics and the existence of vulnerabilities two well-known techniques were used:

- *Pearson correlation* (Benesty et al. [2009]) that evaluates the linear relationship between the software metrics and the existence of vulnerabilities.
- *Spearman correlation* (Myers and Sirois [2006]) that evaluates the monotonic relationship between software metrics and security vulnerabilities.

Although we used both Pearson and Spearman correlation coefficients, it is known that the Pearson correlation is typically used for normally distributed data (data that follow a bivariate normal distribution). For non-normally distributed continuous data, for ordinal data, or for data with relevant outliers, a Spearman rank correlation can be used as a measure of a monotonic association (Schober [2018]). Therefore, Pearson correlation results are not the most adequate for our data, which makes the Spearman correlation results more appropriate to use and discuss.

Table 3.4 presents the project-level metrics that are highly correlated (i.e., Spearman or Pearson correlation $>= 0.9$) with the number of reported vulnerabilities for the five projects in the dataset (the correlation coefficients were calculated using the project-level metrics data of all projects), ordered first by the Spearman and then by the Pearson correlation values.

An interesting observation is that the *Coupling Between Objects (CBO)* (calculated by counting the number of functions/methods of a file/class that are coupled with other files/classes) shows a very strong positive linear and monotonic correlation (i.e., both Spearman and Pearson correlations are high) with the number of vulnerabilities (refer to Figure 3.2 (a)). Thus, a higher CBO not only decreases the software modularity, but also suggests a lower software security level. The *SumEssential* complexity metric (calculated by counting the cyclomatic complexity after iteratively replacing all structured programming primitives with a single statement) also shows a very strong positive monotonic relationship (not linear) with the number of vulnerabilities (refer to Figure 3.2 (b)). This means that the number of vulnerabilities increases with the increasing value of *SumEssential*, but not necessarily at a constant rate.

Table 3.4: Correlated metrics with the number of vulnerabilities in a project.

Software Metrics	Spearman Correlation	Pearson Correlation	Software Metrics Description
Avg CBO	1,00	0,99	Average coupling between objects in each file.
Avg SumEssential	1,00	0,40	Average of the sum of essential complexity of all nested functions or methods in each
Avg CountDeclFunction	0,90	0,53	Average number of functions in each file.
Avg FanOut	0,90	0,47	Average number of called subprograms plus global variables set.
Avg SumMaxNesting	0,90	0,44	Average sum of maximum nesting level of control constructs.
Avg CountStmtDecl	0,90	0,32	Average number of declarative statements in each file.
Avg HK	0,90	0,30	Average HK = (SLOC in the function)* (FanIn * FanOut)^2.
Avg SumCyclomaticModified	0,90	0,27	Average of the sum of modified cyclomatic complexity of all nested functions in each
Avg LCOM	0,70	0,97	Average lack of cohesion in methods of each file.

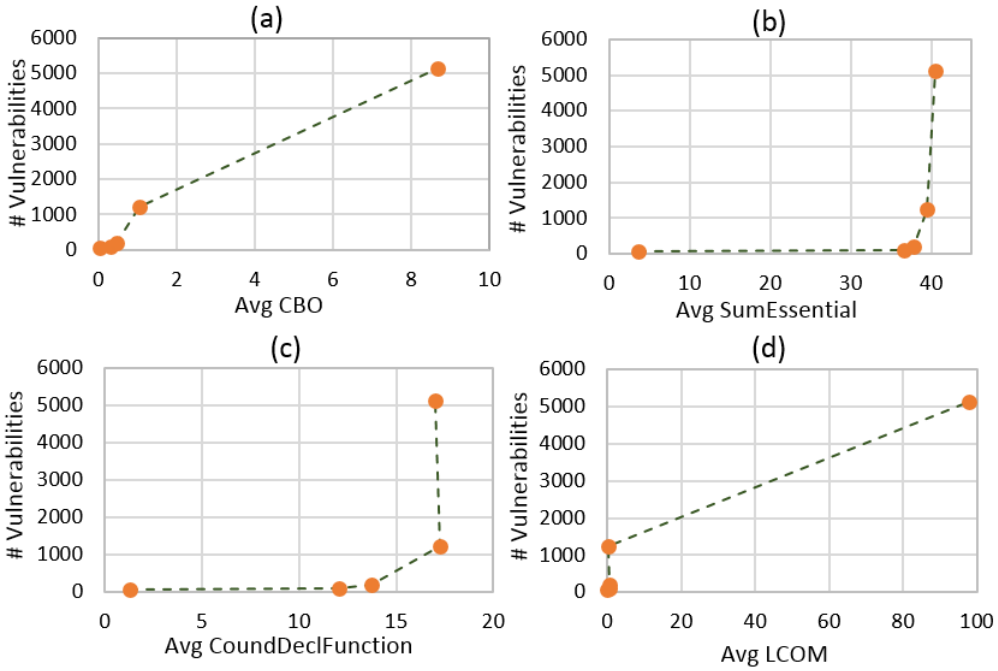


Figure 3.2: Project-level metrics and number of vulnerabilities.

CountDeclFunction (number of functions), *FanOut* (number of called subprograms plus global variables set), *SumMaxNesting* (sum of the maximum nesting level of control constructs like *if* and *while*), *CountStmtDecl* (number of declarative statements), *HK* (information flow relative to function size), and *SumCyclomaticModified* (identical to cyclomatic complexity except that an entire *switch*

statement counts as one) are similarly correlated to the quality of the software in terms of security, all with strong positive monotonic (with Spearman Correlation = 0.9) relations (e.g., refer to Figure 3.2 (c) for the *CountDeclFunction*).

Finally, *Lack of Cohesion (LCOM)* in functions (calculated by removing the number of function pairs that share other class/file fields from the number of function pairs that do not share any field of other class/file) has a strong positive linear connection with the number of vulnerabilities (Pearson Correlation = 0.97). This means that the data is linearly scattered, but not always the number of vulnerabilities increases with the increasing value of *LCOM* (refer to Figure 3.2 (d)).

Observing these results, we can conclude that there is a strong correlation between several project-level metrics and the number of vulnerabilities reported. This means that these metrics are good indicators of software security and may be useful for detecting or indicating vulnerable code.

3.2.2 Dimension Reduction

There are several strategies to deal with the issue of identifying the less-informative software metrics (Feizi-Derakhshi and Ghaemi [2014]), such as:

- *Exponential search*, which is the most exhaustive search technique, guaranteeing that the optimal subset of software metrics is found. Nevertheless, this strategy is not promising or not feasible in practice when the number of features (software metrics in our work) is high (for a feature set of size n, the number of iterations would be 2^n).
- *Heuristic search*, which tries to guarantee the convergence to the (near) best subset of software metrics. This strategy is time consuming and its results depend on the classification model that is used as fitness function.
- *Statistical-based filtering* can be used to find out which metrics *may* not be informative for the detection of vulnerable code units.

Four our study, we selected the last strategy, *Statistical-based filtering*, since it is relatively fast and independent from the classification models.

3.2.2.1 Dimension reduction process

Figure 3.3 presents the process for dimension reduction. As shown, a detailed correlation and a redundancy analysis were conducted on the software metrics at file and function levels, for the five projects included in the dataset. These analyses allow identifying **the least relevant or irrelevant software metrics** (i.e., not or lowly correlated with the class under study, which is the existence of vulnerability), and the **redundant software metrics** (with respect to other metrics).

To identify the **irrelevant metrics**, we calculate the correlation between metrics and the existence of vulnerabilities using *Pearson* (Benesty et al. [2009]) and *Spearman* (Myers and Sirois [2006]) correlation coefficients. In practice, both

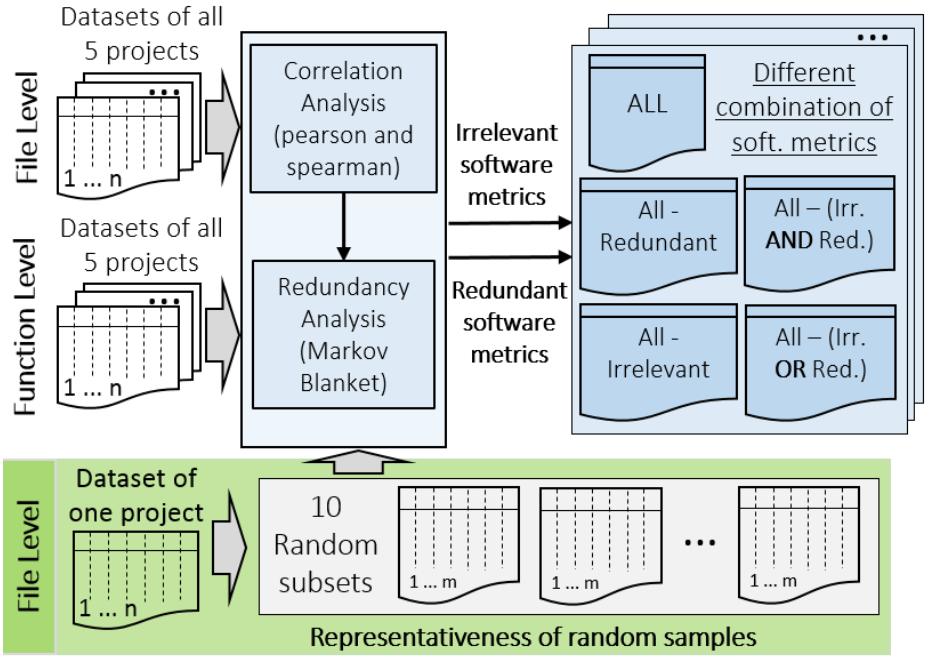


Figure 3.3: Dimension reduction process.

Pearson and Spearman correlation coefficient techniques were used to distinguish highly correlated features (i.e., when value of one feature increases then the value of other feature increases by a consistent amount) from the irrelevant ones.

The software metrics can then be ranked by correlation value (from the highly correlated metrics to the least correlated ones). To select the irrelevant software metrics from this ordered list, a threshold should be defined. In this study, we consider the median as the threshold, as it is commonly used in the literature (Bommert et al. [2020]). In practice, *the software metrics with both Pearson and Spearman correlation values below the median are considered as Irrelevant.*

To identify the **redundant software metrics**, the Markov Blanket Filtering (Yu and Liu [2004]; Wang et al. [2017]) is used. In this filtering technique, let G be the current set of software metrics: if software metric (SM) SM_j has a Markov Blanket SM_i within G, it suggests that SM_j contributes with no more information beyond SM_i to the target class (i.e., existence of vulnerability in this work), and, therefore, SM_j can be safely removed from G. Based on the Approximate Markov blanket definition from Yu and Liu [2004], given two predictive software metrics SM_i and SM_j and the target class V, SM_j is redundant to SM_i, if both equations 4.1 and 4.2 are true:

$$C(SM_i, V) \geq (SM_j, V) \tag{3.1}$$

$$C(SM_i, SM_j) > C(SM_j, V) \tag{3.2}$$

where, $C(SM_i, V)$ is the correlation coefficient between SM_i and the target class

V; $C(SM_j, V)$ is the correlation coefficient between SM_j and the target class V; and $C(SM_i, SM_j)$ is the correlation coefficient between the two predictive software metrics SM_i and SM_j. For this analysis, we again used both Pearson and Spearman techniques to calculate the correlation coefficient. In practice, *we consider software metrics as Redundant when they are identified as so (based on the Approximate Markov blanket), using both Pearson and Spearman techniques.*

After identifying the irrelevant and redundant metrics, we generated 5 groups to be analyzed in further experiments (the goal is to understand whether dimension reduction based on correlation and redundancy analyses can help to achieve better results):

i) **All**, which includes all software metrics present in the dataset;

ii) **All - Irrelevant** that includes all metrics minus the ones that are considered as irrelevant;

iii) **All - Redundant**, which includes all metrics minus the ones that are considered as redundant;

iv) **All - (Irrelevant AND Redundant)** that includes all metrics minus the ones that are listed as irrelevant and as redundant;

v) **All - (Irrelevant OR Redundant)**, including all metrics minus the ones that are listed as irrelevant or as redundant.

In Section 3.3, these different groups of software metrics will be combined with different Machine Learning algorithms in order to analyze the performance of several classification models using different sets of software metrics.

3.2.2.2 Dimension Reduction Results

The statistical correlation analysis to identify the least relevant and the redundant software metrics (to perform dimension reduction) were performed for all projects (Mozilla Firefox, Linux kernel, Xen, Apache and Glibc) at both file and function levels. Tables 3.5 and 3.6 present the results obtained for both levels.

Although the list of irrelevant or redundant metrics identified are not the same in all projects, we can see a high level of similarity between them. For instance, in Table 3.5(a) we can observe that: *(i)* from the 27 file-level metrics (out of a total of 51 metrics) that are considered as *irrelevant* in all five projects, 25 appear at least in 3 projects (e.g., AvgCyclomatic, AltAvgLineBlank, AvgCyclomaticModified, AvgCyclomaticStrict); and *(ii)* similarly, from the 38 file-level metrics considered as *redundant* in all projects, 26 appear at least in 3 projects (e.g., AvgCyclomatic, CountLineBlank, CountLineCodeExe, CountSemicolon). As for the function-level metrics (Table 3.6), we can observe that: *(i)* there is a total of 17 function-level metrics (out of a total of 28 metrics) that are considered as *irrelevant*; and *(ii)* there is a total of 19 function-level metrics (out of a total of 28 metrics) that are considered as *redundant*.

To verify the *Representativeness of Random Samples*, we repeated the above analyses ten times over 10 random samples (with 10000 records each) of file level data of the Mozilla Firefox project. The results, presented in Table 3.5(b), show that, in all cases, 30 software metrics (out of a total of 51 file level metrics) are identified as irrelevant and 28 software metrics are identified as redundant. In addition to that, there is a large group of metrics that appear repeatedly across different sample sets as irrelevant and redundant. For example, as shown in the last two columns of Table 3.5, 26 out of 30 irrelevant metrics are identified in at least 7 samples (e.g., FanOut in 10 samples, AvgCyclomatic in 9 samples). We obtained similar results regarding redundant metrics: out of a total of 28 redundant metrics, 23 are identified as such in at least 6 samples (e.g., AvgCyclomatic in 10 samples, AltAvgLineBlank in 7 samples).

Table 3.5: Irrelevant and redundant file-level software metrics (a) and their frequency in 10 random samples of Mozilla Firefox (b).

	(a) Irrelevant (I) and Redundant (R) File level metrics						(b) # of samples	
#	Software Metrics	MOZILLA	KERNEL	XEN	APACHE	GLIBC	Irr.	Red.
1	AvgCyclomatic	I / R	I / R	I / R	I / R	I / R	9	10
2	AltAvgLineBlank	I / R	I / R	I / R	I	R	9	7
3	AvgCyclomaticModified	I / R	I	I	I / R	I / R	8	7
4	AvgCyclomaticStrict	I	I / R	I	I / R	I / R	8	1
5	AvgLine	I	I / R	R	I	I / R	9	0
6	FanOut	I / R	I	I / R	I / R	I	10	1
7	FanIn	I	I	I	I	I / R	10	0
8	SumMaxNesting	I	I	I	I	I / R	10	0
9	MaxMaxNesting	I	I	I	I	R	10	0
10	HK	I	I	I	I	I	9	0
11	LCOM	I	I	I	I	I	9	0
12	MaxFanIn	I	I	I	I	I	10	0
13	CountPath	I	I	I	I	I	10	0
14	CBO	I	I	I	I	I	10	0
15	CountLineBlank	R	R	R	R	R	0	9
16	CountLineCodeExe	R	R	R	R	R	0	10
17	CountSemicolon	R	R	R	R	R	0	10
18	CountStmt	R	R	R	R	R	0	10
19	CountStmtExe	R	R	R	R	R	0	10
20	MaxCyclomatic	R	R	R	R	R	1	9
21	MaxCyclomaticModified	R	R	R	R	R	1	6
22	AltAvgLineCode	I / R	I / R		I / R	I / R	10	9
23	AvgLineCode	I / R		R	I / R	I / R	10	8
24	AvgLineBlank	I	I	I	I / R		9	1
25	CountLineCode		R	R	R	R	0	7
26	SumCyclomatic	R	R	R		R	0	9
27	SumCyclomaticModified	R	R	R	R	R	0	8
28	AltCountLineCode	R	R		R	R	0	9
29	AltCountLineComment	R	R	R		R	0	9
30	SumCyclomaticStrict	R		R	R	R	0	8
31	RatioCommentToCode	I	I		I	I	10	0
32	CountStmtEmpty	I	I	I		I	7	0
33	AvgFanIn	I	I	I	I		10	0
34	AltAvgLineComment	I / R	I / R	I			9	10
35	AvgEssential	I			I	I / R	8	0
36	AvgLineComment	I	I	I / R			8	0
37	AltCountLineBlank		R	R	R		0	6
38	CountDeclFunction		R	R		R	1	0
39	CountLine	R		R		R	0	8
40	CountLineCodeDecl	R		R	R		0	10
41	CountStmtDecl		R	R	R		0	0
42	CountLineInactive		I	I	I		2	0
43	MaxFanOut	I		I	I		10	0
44	AvgMaxNesting	I				I / R	7	0
45	MaxCyclomaticStrict			R		R	0	2
46	AvgFanOut					R	8	0
47	CountLineComment					R	0	2
48	SumEssential				R		0	6
49	CountLinePreprocessor				I		0	0
50	MaxEssential						0	0
51	MaxNesting						0	0

The results suggest that the random samples have quite similar characteristics and patterns in terms of correlation between the software metrics, and between the software metrics and the existence of vulnerabilities, which are important factors in building predictive models out of software metrics. One of these samples was randomly chosen for further experiments and analysis (in Section 3.3) to ensure the avoidance of any sampling bias that may exist.

Table 3.6: Irrelevant and redundant function-level software metrics.

Irrelevant (I) and Redundant (R) Function level metrics						
#	**Software Metrics**	**MOZILLA**	**KERNEL**	**XEN**	**APACHE**	**GLIBC**
1	MinEssentialKnots	I / R	I / R	I / R	I / R	R
2	MaxEssentialKnots	I / R	I / R	I	I / R	I / R
3	CyclomaticStrict	R	R	R	I / R	R
4	AltCountLineBlank	R	R	I / R	R	R
5	CountStmtExe	R	R	R	R	R
6	Cyclomatic	R	R	R	R	R
7	CountLineCodeExe	R	R	R	R	R
8	CountLineInactive	I	I	I	I	I
9	CountLinePreprocessor	I	I	I	I	I
10	CountStmtEmpty	I	I	I	I	I
11	AltCountLineCode	R	R		R	R
12	CyclomaticModified	R	R	R	R	
13	CountSemicolon	R		R	R	R
14	CountStmt	R	R	R		R
15	AltCountLineComment	R	R	I	I	R
16	RatioCommentToCode		I	I	I	I
17	CountLine	R			R	R
18	CountLineCode	R		R	R	
19	Knots	I	I			I
20	Essential	I		I / R	I	
21	CountLineBlank		R	I / R		
22	CountLineCodeDecl				I	I / R
23	CountLineComment			I / R	I	
24	CountStmtDecl			I		R
25	CountInput					I
26	MaxNesting					I
27	CountOutput					
28	CountPath					

3.2.3 Feature selection

A feature selection technique, namely, a genetic algorithm combined with the Random Forest learning algorithm, was used to select the most predictive software metrics to distinguish vulnerable from non-vulnerable code units. In the following subsections, we describe and detail the entire feature selection procedure, which is depicted in Figure 3.4. Also, we present the genetic algorithm and its configuration process that includes: *(i)* a calibration step to choose the best possible values for the parameters, and *(ii)* a dataset size setting to identify which data size is the most adequate to achieve a high quality solution in a reasonable amount of time. Finally, the results of the feature selection process are discussed (file and function level), as well as the analysis carried out to verify the effectiveness of the feature selection results.

3.2.3.1 Feature Selection process

Figure 3.4 presents a very well-known feature selection process (Guyon and Elisseeff [2003]), that was adapted to our study and includes four stages:

1. **Search or Generation of Subset**: consists of searching for a new subset of metrics from the original set of metrics in the dataset. This can be done in four different ways: *i)* the search may start with an empty set and

successively, in each iteration, new metrics are added (*Forward Search*); *ii)* it can be started with the full set of metrics and then some of them are consecutively eliminated in each iteration (*Backward Elimination Search*); *iii)* it can be done by simultaneously adding and removing metrics (*Bi-directional Search*); or *iv)* it can begin with a random subset and continue by randomly selecting or eliminating metrics (*Random Search*). We use the later approach mainly due to the fact that the use of randomness helps avoiding trapping into local optima in the search space (Srinivas and Patnaik [1994]).

In addition to the search direction, we need to specify the search strategy. The basic search strategy is called *Exponential Search*, which is the most exhaustive one, guaranteeing that the optimal subset is found. However, this strategy is not promising when the number of features (software metrics in our study) is high (i.e., for a feature set of size n, the number of iterations would be 2^n). For this reason, we use a heuristic search technique based on a genetic algorithm (Pham and Karaboga [2012]). As shown in Sexton et al. [1999] and Braun et al. [2001], genetic algorithms not only guarantee convergence to the (near) best solution, but also offer a (relatively) rapid convergence and high computational efficiency. Details on the use of the genetic algorithm are provided in the Section 3.2.3.2.

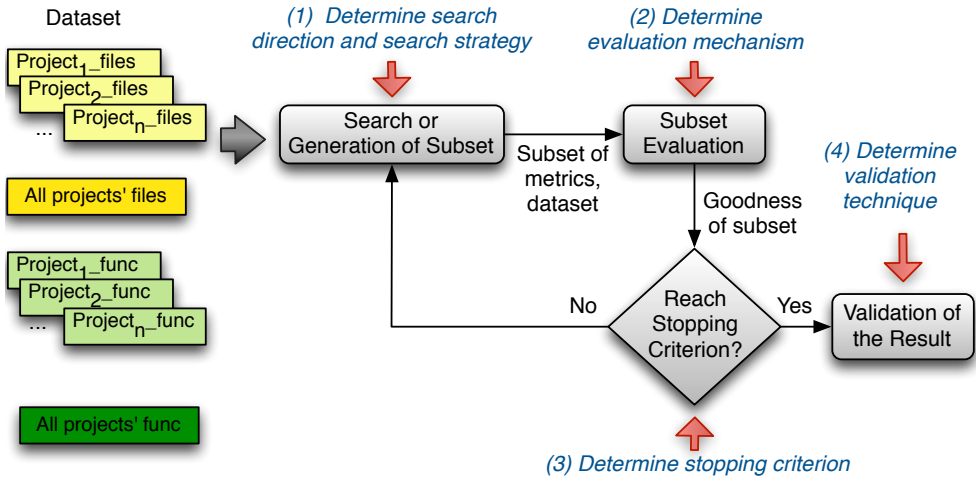


Figure 3.4: Feature selection procedure.

2. **Subset Evaluation**: the generated (or selected) subset of metrics is evaluated. To accomplish this task, we use a supervised wrapper technique (Guyon and Elisseeff [2003]), as evaluation approach, which builds a predictive classifier model based on a labeled dataset composed of the chosen subset of metrics. The quality of this model exposes the goodness of the subset. In practice, we use **Accuracy**, which is the most common criterion for evaluating classifier models (Witten et al. [2016]), to measure the quality of the model in each iteration of the genetic algorithm, thus the best subset of metrics is selected by maximizing the Accuracy value. The Accuracy criterion, representing the proportion between the correctly classified code units (e.g., functions) and the total number of code units,

is calculated using Equation 3.3, where TP, TN, FP, and FN respectively stand for: true positives, true negatives, false positives, and false negatives.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.3}$$

In addition to accuracy, we also use Cohen's kappa (Wood [2007]), as a complementary criterion when Accuracy has the same value in two different cases, which shows how much better or worse our classifier model is compared to what it would be expected by random chance. This criterion is calculated using Equation 3.4.

$$Kappa = \frac{Accuracy - ExpectedAccuracy}{1 - ExpectedAccuracy} \tag{3.4}$$

To build the predictive classifier, we use Random Forest (Breiman [2001]), which is one of the most popular ensemble learning algorithms. It consists of a combination of several decision tree classifiers, each one fitted on a random sub-sample of a dataset, making it more accurate and robust to outliers and noise than a single classifier (Breiman [2001]). The reason for choosing the Random Forest is two-fold: *i)* it is accurate and does not overfit, so results are general enough to be extended to other classifiers; and *ii)* it efficiently runs on a large dataset with a large set of features (Breiman [2001]).

3. **Reach Stopping Criterion?**: either the search process finishes or starts all over again from the beginning, depending on the stopping criterion. Since we are using a genetic algorithm, the maximum number of iterations (or generations) is considered as the stopping criterion. When the value of the stopping criterion (number of iterations) reaches a predefined value (150 for file-level and 125 for function-level analysis, which are experimentally observed to be enough for convergence (see Section 3.2.3.2)), it stops and provides the (near) best subset of metrics.

4. **Validation of the Result**: the results obtained by the genetic algorithm combined with the Random Forest classifier, are validated. To validate the result of this heuristic search, we perform a convergence analysis to see whether the values assigned to the genetic algorithm parameters lead to a convergence to the (near) best result. To do so, the algorithm is executed several times over the same dataset in order to verify the similarity of the results. Based on evidences gathered from the literature (Grefenstette [2012]), we assume that the results of genetic algorithms are reliable if the the convergence rate is more than 60%.

3.2.3.2 The Genetic Algorithm

The previously presented stages are performed by applying the genetic algorithm as follows:

1. The genetic algorithm starts on the basis of an initial population containing a set of chromosomes (i.e., candidate solutions). In our case, a set including several subsets of metrics, which are randomly generated, represents the initial population in the genetic algorithm.
2. A fitness value (Accuracy) is calculated by the random forest classification algorithm and assigned to each individual of the population (each subset of metrics).
3. The best individuals (with better fitness values) of the current population are randomly combined by employing both crossover and mutation operations to produce the population of the next generation.
4. The previous steps are repeated over and over until the maximum number of generations has been achieved.

This heuristic search, which uses the genetic algorithm combined with the random forest learning algorithm, is time consuming. There are several important parameters in the genetic algorithm, whose values may influence both the search result and the execution time. These parameters are *population size*, *number of generations*, *crossover probability*, and *mutation probability*. Selecting the right values for the genetic algorithm parameters is a difficult context-dependent problem that needs to be addressed adequately. Otherwise, high-quality solutions are unlikely to be found in reasonable time. In addition to these parameters, the size of the data used in the random forest learning algorithm influences the training time and the performance of the classifier model. In fact, there is a trade-off between the performance of the search algorithm and the time taken to find the solution. Usually, a better solution is found when more data is used for building the classifier model and more generations are produced by the genetic algorithm. However, these conditions impose more time for the search. Based on this, a preliminary analysis was conducted addressing two aspects:

- A **calibration of the Genetic Algorithm** to choose the best possible values for the parameters;
- A **dataset size setting** considering multiple data sizes, to identify which one is the most adequate to achieve a high quality solution in a reasonable amount of time.

Calibration of the Genetic Algorithm: To choose the best possible values for the genetic algorithm parameters, the algorithm was calibrated as follows: *i)* a set of values for each parameter based on evidences gathered from the literature (Grefenstette [2012]) was defined; *ii)* the genetic algorithm was executed by setting all the possible combinations of these values; and *iii)* the best combination (i.e. the one that led to the highest accuracy) was selected.

The algorithm was run over the data of the Apache httpd project. Since this

empirical approach for calibrating the genetic algorithm is very time consuming and the size of the data strongly and linearly influences that time, the Apache httpd was selected, which is the smallest project in the dataset (see Table 3.3 in Section 3.1). Note that, the calibration results depend on the given problem instance and dataset (De Landgraaf et al. [2007]) and due to the fact that the projects used in this work are different in terms of functionality and code structure, the algorithm configured based on the result of this calibration may not lead to the best accuracy in all projects. Nevertheless, the calibration process allows us to achieve our objective (i.e., to demonstrate that it is possible to identify the most predictive software metrics to distinguish vulnerable from non-vulnerable code units).

The initial test values defined for each parameter are as follows:

- Population size <- (30, 40, 50)
- Number of generations <- (100, 125, 150)
- Crossover probability <- (0.7, 0.8, 0.9)
- Mutation probability <- (0.1, 0.2, 0.3)

Considering the **file-level metrics**, a total of 81 experiments was performed by setting all the possible combinations of the values. From all these, we highlight in Table 3.7 the six best results in terms of accuracy using *file-level metrics* of Apache httpd. To choose the best of the bests, we used the Cohen's kappa criterion. The results show that the combination with population size = 40, number of generations = 150, crossover probability = 0.8 and mutation probability = 0.3, leads to the best result.

Table 3.7: The top 6 best results of calibration using file-level metrics.

Population Size	# Generations	Crossover Prob.	Mutation Prob.	Accuracy	Kappa
40	**150**	**0,8**	**0,3**	**0,9778**	**0,5192**
40	150	0,7	0,3	0,9778	0,5041
30	150	0,7	0,1	0,9778	0,5041
30	150	0,8	0,3	0,9778	0,4835
30	150	0,8	0,1	0,9778	0,4818
30	125	0,7	0,2	0,9778	0,455

To validate the results above, we repeated the search algorithm with the selected parameters and the same dataset to understand how similar the outcomes are. The results (presented in Table 3.8) show that in 7 cases out of 10 (70% of times), the algorithm converged to the best Accuracy (0.9766). The list of metrics selected was exactly the same in the cases with the same Accuracy and Kappa. Based on evidences in the literature (Grefenstette [2012]), this results is a good indication of the genetic algorithm convergence.

We repeated the calibration process for the Apache httpd's **function-level metrics**. From a total of 81 combinations of the values defined for each genetic

Table 3.8: Validation of the genetic algorithm parameters.

	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Exp7	Exp8	Exp9	Exp10
Accuracy	0,9753	0,9766	0,9766	0,9766	0,9753	0,9766	0,9766	0,9766	0,9753	0,9766
Kappa	0,4398	0,4515	0,4889	0,4514	0,4398	0,4515	0,4889	0,4514	0,4398	0,4515

algorithm parameter, the results showed that the combination with population size = 50, number of generations = 125, crossover probability = 0.7 and mutation probability = 0.3, leads to the best result in both Accuracy (0.9730) and Kappa (0.3703), as we can observe in Table 3.9.

Table 3.9: The top 3 best results of calibration using function-level metrics.

Population Size	# Generations	Crossover Prob.	Mutation Prob.	Accuracy	Kappa
50	**125**	**0,7**	**0,3**	**0,9730**	**0,3703**
50	100	0,9	0,2	0,9720	0,3096
30	100	0,9	0,2	0,9720	0,2894

Dataset size setting: To understand which dataset size is enough to get a high quality result in a reasonable time, we run the search algorithm over datasets of different (linearly increasing) sizes. For this, we used the data for Mozilla Firefox's files (one of the biggest projects) to ensure that there was no strict limitation for increasing the size, as much as necessary, until finding the reasonably best size (a dataset from a small project would limit the analysis).

Since the number of samples regarding vulnerabilities is small in all projects, we decided to keep all these for the tests. This way, we started the experiment by randomly selecting 830 records without vulnerabilities and then added 830 records with vulnerabilities (i.e. all the vulnerabilities for the Mozilla Firefox project). We then created additional datasets by increasing the size of non-vulnerable data linearly, until the accuracy results stabilized.

Figure 3.5(a) presents the datasets considered and the combination between their vulnerable and non-vulnerable records.

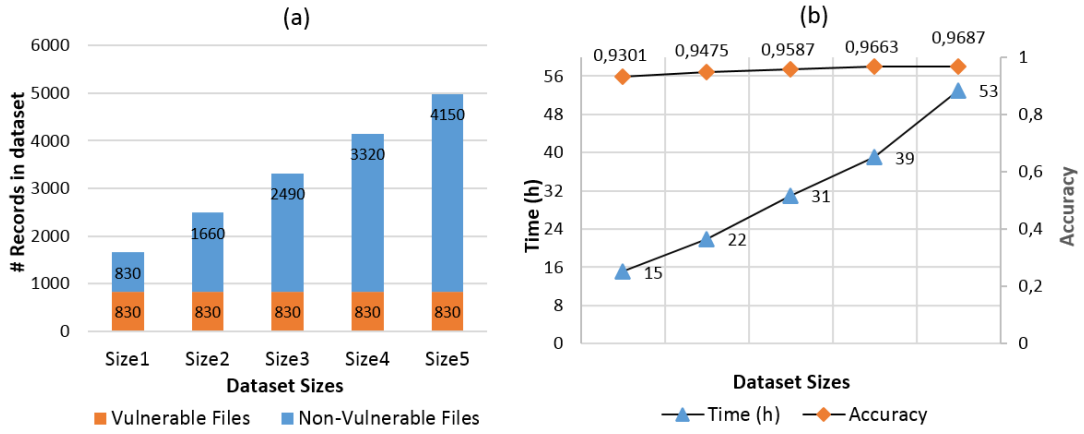


Figure 3.5: Analysis of data size over Mozilla Firefox files.

Figure 3.5(b) presents the accuracy achieved in each case (scale on right side vertical axis) and the time taken to achieve this results (scale on the left side vertical axis). As shown, accuracy increases, although smoothly, for increasing sizes of data. As for the time taken by the algorithm, an almost linear increase can be observed. When the data size increases from 4150 to 4980, we see a huge increase in time (from 39 hours to 53 hours), while the increase in accuracy is low (from 0.9663 to 0.9687). For this reason, we chose 4150 as the size of the dataset to be used in this experiment (feature selection using the genetic algorithm). For the smaller projects (e.g., Apache httpd), whose number of files is less than 4150, we used the whole dataset in all cases. Although this analysis could lead to slightly different results if we considered other projects rather than Mozilla Firefox, it is good enough for our purposes as using this number of records allows us to keep all vulnerable files or functions plus at-least the equal number of non-vulnerable ones in all cases.

3.2.3.3 Assess the effectiveness of the feature selection

The genetic algorithm was applied using file-level and function-level software metrics over the data of the five projects (as will be discussed in sections 3.2.3.4 and 3.2.3.5). To assess the effectiveness of the feature selection process in each case, we run the genetic algorithm and the Random Forest classifier for all projects using the configurations (calibrations of parameters and dataset size) identified previously. When the stopping criterion (number of iterations) reaches a predefined value, the process stops and the the subset of software metrics that allow achieving the highest accuracy is selected.

To demonstrate the effectiveness of the selected subsets of metrics for each project, we compare the accuracy of the classifier model built using that subset with the accuracy of models built using:

i) the top correlated metric;

ii) the top 10 individually correlated metrics;

iii) the intersection of the metrics selected for all projects;

iv) the intersection of the metrics selected for at least three projects;

v) all metrics.

To select the top correlated metrics, we calculate the correlation between the software metrics and the existence of vulnerabilities in functions and files by using the Point-Biserial correlation coefficient (Tate [1954]), which is suitable when one variable (existence of vulnerabilities, in this case) is dichotomous. Furthermore, we cross-validate the results obtained for each project, by using the datasets from the other projects.

3.2.3.4 File-level feature selection results

Table 3.10 presents the results obtained from the heuristic search for the five projects individually and collectively (i.e., combination of the data from the five

projects) considering the file-level metrics. The table presents the Accuracy and Kappa of the selected subsets, the number of records included in the dataset, and the number of vulnerable records used for each project. In general, the number of the metrics selected and their combination varies from one project to another (e.g., from 12 metrics in Glibc to 21 metrics in Mozilla Firefox), but there are similarities between the selected subsets of metrics.

Table 3.10: File-level results for each project.

Projects	Apache httpd	Xen HV	Glibc	Linux Kernel	Mozilla Firefox	All Projects
# Records in Dataset	810	1257	865	4150	4150	4150
# Vulnerable Records	29	71	31	842	830	1803
Accuracy	0,9778	0,9761	0,9664	0,922	0,9665	0,9116
Kappa	0,5192	0,6975	0,294	0,7247	0,889	0,818
# Selected Features	20	17	12	16	21	16

These similarities are shown in Table 3.11 in different shades of green: the darker green indicates that the metric is selected in more projects.

Table 3.11: File-level metrics selected by the heuristic search for each project.

	Apache httpd	Xen HV	Glibc	Mozilla Firefox	Linux Kernel	All Projects
Selected Features	AvgMaxNesting	AvgMaxNesting	AvgMaxNesting	AvgMaxNesting	AvgMaxNesting	AvgMaxNesting
	SumMaxNesting	SumMaxNesting		SumMaxNesting	SumMaxNesting	SumMaxNesting
	MaxMaxNesting	CountPath		CountPath	CountPath	CountPath
	MaxNesting	FanIn		FanIn	FanIn	FanIn
	AvgFanIn	MaxFanIn	AvgFanIn	CountStmtEmpty	AvgFanIn	AvgFanIn
	MaxFanOut			FanOut	FanOut	FanOut
	AvgFanOut	AvgFanOut		AvgFanOut	AvgFanOut	AvgFanOut
	CountLineComment	AvgEssential	CountLineComment	AvgEssential		
	AltCountLineCode	SumEssential	SumEssential	CountLineCodeDecl	SumEssential	SumEssential
	SumCyclomatic Modified	AltCountLine Comment		SumCyclomatic Modified	SumCyclomatic Modified	AltCountLine Comment
	MaxCyclomatic	AvgLineComment	MaxCyclomatic	AvgCyclomatic Modified	AvgCyclomatic	AvgCyclomatic
	CountLineInactive		CountLineInactive	SumCyclomatic	SumCyclomatic	
	CountLineCode	CountStmt	CountStmt	AvgCyclomaticStrict	CountLineCode	
	SumCyclomaticStrict		SumCyclomaticStrict	SumCyclomaticStrict	AltAvgLineCode	AltAvgLineCode
	CBO	CBO		CBO		CBO
	LCOM	CountLineBlank		LCOM		CountLineBlank
	AltCountLineBlank	AltCountLineBlank		CountSemicolon	AltCountLineBlank	CountSemicolon
	CountStmtExe		CountStmtExe	AvgLineCode	AvgLineCode	AvgLineCode
	CountLineCodeExe	CountDeclFunction	CountDeclFunction	CountDeclFunction	CountDeclFunction	CountDeclFunction
	AltAvgLineBlank		AltAvgLineBlank	AltAvgLineBlank		
		RatioCommentTo Code	RatioCommentTo Code	RatioCommentTo Code		

By analyzing these results, we first observe that all metrics selected in the combined dataset (data from all projects; last column in Table 3.11) are also selected in at least one project, but the reverse is not true. For instance, *LCOM* that is selected in Apache httpd and Mozilla Firefox, is not selected in the case where all data is combined. This is mainly due to the fact that the projects are

architecturally different from each other; therefore, the combination of their architectural information may diminish the importance of some metrics that were discriminative in one of the projects. The fact that all metrics that are selected in four and five projects (e.g., *AvgMaxNesting* and *CountDeclFunction*) are also selected in the combined dataset, supports this justification. *AvgMaxNesting*, the average of maximum nesting level of control constructs in functions of a file, is a clear example of that.

We also observe that the selected metrics are from all groups, including complexity (e.g., SumEssential), volume (e.g., CountDeclFunction), coupling (e.g., CBO), and cohesion (e.g., LCOM). This exposes the limitation of previous research that focus only on complexity metrics (Shin and Williams [2011]; Shin [2008]; Shin and Williams [2008]) for detecting vulnerabilities or for improving software security.

We further observe that, from the metrics showing high correlation with the number of vulnerabilities at the project-level (refer to Table 3.4), the first five (*CBO*, *SumEssential*, *CountDeclFunction*, *FanOut*, and *SumMaxNesting*) are present in the subset of metrics selected in the combined dataset. *HK* and *CountStmtDecl* (number of declarative statements) are not present in the resulting subset of any project, but instead we can find some metrics that are highly correlated (Spearman and Pearson correlation >= 0.9) with these two metrics. In particular, *FanIn* and *FanOut*, which are highly correlated with *HK*, and *CountStmt* (Number of statements), which is highly correlated with *CountStmtDecl*, are selected in various projects.

The last observation is that, in the cases where the number of records with vulnerabilities is higher in a dataset (i.e., more vulnerabilities are reported), we get higher Kappa showing that the classifier model is more precise. The same is not observed for Accuracy, because the proportion of vulnerable and non-vulnerable records are not necessarily the same in the datasets, so the number of non-vulnerable records classified as non-vulnerable (true negatives) influences the accuracy value. Thus, we see that accuracy is usually higher for datasets with a lower number of vulnerable records (e.g., 0.9778 in Apache httpd and 0.922 in Linux Kernel), but that does not mean that the classifier model is more precise. For this reason, we cannot use the accuracy value to compare the results obtained for different datasets of different projects with different proportions of vulnerable and non-vulnerable records.

The Kappa value, based on Viera et al. [2005], is interpreted as follows: *poor* when less than 0, *slight* between 0.01 and 0.2, *fair* between 0.21 and 0.4, *moderate* between 0.41 and 0.6, *substantial* between 0.61 and 0.8, and finally *almost perfect* between 0.81 and 0.99. Based on this, we can see that we could build substantial and close to perfect classifiers using the selected metrics in the case of all projects, except for Glibc.

To **validate the effectiveness of the selected subsets of metrics**, we compare them with several relevant individual or groups of metrics. As shown in Tables 3.12 and 3.13, we compare the accuracy of the classifier model built using the subset of metrics selected by the genetic algorithm with the accuracy of the

Table 3.12: Comparison between the accuracy of the selected file-level metrics of Apache httpd, Xen HV and Glibc projects with several subsets.

	Apache httpd			Xen HV			Glibc		
	Metrics	**Accuracy**	**Kappa**	**Metrics**	**Accuracy**	**Kappa**	**Metrics**	**Accuracy**	**Kappa**
1 Top Correlated Metric	AvgFanOut	0,9703	0,2434	AvgFanOut	0,9681	0,6004	AltCountLineBlank	0,9535	-0,0199
10 Top Correlated Metric	AvgFanOut HK SumCyclomaticModified CountSemicolon AvgMaxNesting CountStmt CountStmtExe CountDeclFunction SumCyclomaticStrict SumEssential	0,9703	0,3873	AvgFanOut SumEssential SumCyclomaticStrict CountStmtExe SumCyclomaticModified SumCyclomatic CountStmt CountLineBlank CountSemicolon AltCountLineBlank	0,9649	0,5755	AltCountLineBlank CountLineBlank SumEssential MaxEssential CountLinePreprocessor CountStmtDecl AltAvgLineComment AvgLineComment AvgLine AltAvgLineCode	0,9767	0,4363
Selected metrics in all projects	AvgMaxNesting	0,9703	0,3873	AvgMaxNesting	0,9712	0,6270	AvgMaxNesting	0,9684	0,2430
Metrics Selected in Most of Projects (3 and more)	AvgMaxNesting SumMaxNesting CountPath FanIn AvgFanIn FanOut AvgFanOut SumEssential SumCyclomaticModified SumCyclomaticStrict CBO AltCountLineBlank CountDeclFunction AltAvgLineBlank RatioCommentToCode	0,9703	0,3873	AvgMaxNesting SumMaxNesting CountPath FanIn AvgFanIn FanOut AvgFanOut SumEssential SumCyclomaticModified SumCyclomaticStrict CBO AltCountLineBlank CountDeclFunction AltAvgLineBlank RatioCommentToCode	0,9744	0,7202	AvgMaxNesting SumMaxNesting CountPath FanIn AvgFanIn FanOut AvgFanOut SumEssential SumCyclomaticModified SumCyclomaticStrict CBO AltCountLineBlank CountDeclFunction AltAvgLineBlank RatioCommentToCode	0,9674	0,0000
All metrics	-	0,9703	0,3873	-	0,9681	0,6269	-	0,9721	0,2438
Selected Metrics by GA	-	0,9778	0,5192	-	0,9761	0,6975	-	0,9664	0,2940

classifier model built using the combinations of metrics mentioned in Section 3.2.3.3.

The results show that, in all projects except Glibc, the Accuracy and Kappa are higher when the metrics selected by the genetic algorithm are used to build the classifier model. To double-check the results of Glibc, we repeated the experiment for this project and achieved a similar result. This means that the genetic algorithm was not able to converge to the (near) best result in the case of Glibc. Since we used the Apache https's data for calibration of the genetic algorithm, the main reason for this situation can be the non-optimal parameter set.

The validation results also show that, in the cases where the top one correlated metric (i.e., correlated with the existence of vulnerabilities) is chosen for building the classifier model, the results are usually worse in terms of Accuracy and Kappa (i.e., here we can use Accuracy for comparison because the dataset used for each project is the same in all cases). This observation exposes the limitations of previous research, which focused on using the correlation between each individual metric with the existence of vulnerabilities (Shin and Williams [2008]; Alves et al. [2016b]) to identify the discriminative and predictive software metrics for vulnerabilities.

Table 3.13: Comparison between the accuracy of the selected file-level metrics of Mozilla Firefox and Linux Kernel projects with several subsets.

	Mozilla Firefox			Linux Kernel		
	Metrics	**Accuracy**	**Kappa**	**Metrics**	**Accuracy**	**Kappa**
1 Top Correlated Metric	CountLineBlank	0,8216	0,3226	AvgFanOut	0,8014	0,2801
10 Top Correlated Metric	CountLineBlank AltCountLineBlank SumEssential CountDeclFunction SumCyclomaticStrict SumCyclomatic SumCyclomaticModified CountLineCode CountStmtDecl AltCountLineCode	0,8341	0,3688	AvgFanOut SumEssential CountStmtExe MaxEssential CountSemicolon CountDeclFunction AltCountLineBlank CountLine CountLineBlank AltCountLineCode	0,8698	0,5350
Selected metrics in all projects	AvgMaxNesting	0,8222	0,1918	AvgMaxNesting	0,8023	0,0533
Metrics Selected in Most of Projects (3 and more)	AvgMaxNesting SumMaxNesting CountPath FanIn AvgFanIn FanOut AvgFanOut SumEssential SumCyclomaticModified SumCyclomaticStrict CBO AltCountLineBlank CountDeclFunction AltAvgLineBlank RatioCommentToCode	0,9643	0,8816	AvgMaxNesting SumMaxNesting CountPath FanIn AvgFanIn FanOut AvgFanOut SumEssential SumCyclomaticModified SumCyclomaticStrict CBO AltCountLineBlank CountDeclFunction AltAvgLineBlank RatioCommentToCode	0,9142	0,7008
All metrics	-	0,9585	0,8603	-	0,9094	0,6715
Selected Metrics by GA	-	0,9665	0,8890	-	0,9220	0,7247

A cross-validation was also performed to see how effective the metrics selected for a given project might be in the other projects. The results, presented in Table 3.14, are consistent with the previous results, as for each project, except for Glibc (shown in red), the metrics selected by the genetic algorithm for that specific project lead to the best results in terms of Accuracy and Kappa (shown in green).

Table 3.14: Cross-validation of the selected file-level metrics.

		Applied to									
		Apache httpd		**Xen HV**		**Glibc**		**Mozilla Firefox**		**Linux Kernel**	
Metrics of	**Apache httpd**	0,9778	0,5192	0,9712	0,6750	0,9923	0,1967	0,9565	0,8549	0,8901	0,6175
	Xen HV	0,9703	0,3873	0,9761	0,6975	0,9628	-0,0082	0,9604	0,8688	0,8939	0,6188
	Glibc	0,9703	0,3873	0,9681	0,6269	0,9664	0,2940	0,9546	0,8496	0,8852	0,5744
	Mozilla Firefox	0,9703	0,3873	0,9681	0,6269	0,9674	0,0000	0,9665	0,8890	0,9103	0,6770
	Linux Kernel	0,9703	0,3873	0,9712	0,6750	0,9674	0,0000	0,9610	0,9042	0,9220	0,7247

3.2.3.5 Function-level feature selection results

Table 3.15 presents the results obtained from the heuristic search for the five projects individually and collectively (i.e., combination of the data from the five projects) considering the function-level metrics. The table presents the Accuracy

Table 3.15: Function-level results for each project.

Projects	Apache httpd	Xen HV	Glibc	Mozilla Firefox	Linux Kernel	All Projects
# Records in Dataset	1000	1000	1000	4214	4214	7100
# Vulnerable Records	33	154	16	2107	1229	3539
Accuracy	0,973	0,87	0,98502	0,7432	0,757	0,7214
Kappa	0,3703	0,353	0,09898	0,4865	0,3323	0,4429
# Selected Features	13	14	13	15	15	16

and Kappa of the selected subsets, the number of records included in the dataset, and the number of vulnerable records used for each project.

Table 3.16 presents the function-level metrics selected for the five projects individually and collectively. As shown, despite being different in number and combination of metrics, the selected subsets of metrics share similarities. Metrics *CountStmtEmpty* (i.e., number of empty statements) and *CountLineCode* (i.e., number of executable lines of source code, also known as *SLOC*) are selected for all projects. The *CountStmtEmpty*, as expected, is also selected for the combined dataset. The *CountLineCode* is not selected but instead, a redundant metric, called *AltCountLIneCode* (i.e., number of lines containing source code, including inactive regions), which is highly correlated with *CountLineCode* (i.e., in all projects the correlation between these two metrics is about 0.99 for both Pearson and Spearman correlation coefficients) is selected.

Table 3.16: Function-level metrics selected by the heuristic search for each project.

	Apache httpd	Xen HV	Glibc	Mozilla Firefox	Linux Kernel	All Projects
Selected Features	CountStmtEmpty	CountStmtEmpty	CountStmtEmpty	CountStmtEmpty	CountStmtEmpty	CountStmtEmpty
	CountLineCode	CountLineCode	CountLineCode	CountLineCode	CountLineCode	**AltCountLineCode**
	CountLineComment	CountLineComment	CountLineComment	CountLineComment		CountLineComment
	CountStmt	CountStmt	CountStmt		CountStmt	CountStmt
	Cyclomatic	CountLineInactive	CountLineInactive	CountLineInactive	CountLineInactive	Cyclomatic
		CountLineCodeDecl	CountLineCodeDecl		CountLineCodeDecl	CountLineCodeDecl
	CountLine Preprocessor	CountLine Preprocessor			CountLine Preprocessor	
		CountSemicolon	CountSemicolon	CountSemicolon		CountSemicolon
	CountStmtDecl			CountStmtDecl	CountStmtDecl	
	CountStmtExe		CountStmtExe	CountStmtExe		CountStmtExe
	CyclomaticModified	AltCountLine Comment	AltCountLine Comment	CyclomaticModified	CyclomaticModified	CyclomaticModified
	Essential	Essential	CountPath	Essential	CountPath	CountPath
	MaxEssentialKnots		MaxEssentialKnots	MaxEssentialKnots		
	RatioCommentTo Code	RatioCommentTo Code	RatioCommentTo Code	CountLine	MaxNesting	RatioCommentTo Code
		CountLineBlank	CountInput		CountInput	CountInput
	CyclomaticStrict	CyclomaticStrict		CountOutput	CountOutput	CountOutput
		CountLineCodeExe		CountLineCodeExe		CountLineCodeExe
				Knots	Knots	Knots
				MinEssentialKnots	MinEssentialKnots	MinEssentialKnots

We also observe that Kappa values are low in comparison to the file-level results. In general, fair and moderate classifiers are built using the selected metrics.

Glibc shows the worse result than the other projects in both file and function levels.

We further observe that volume metrics (e.g., CountStmtEmpty, CountLineCode, CountLineComment, CountStmt) are the majority (i.e., being in four or five projects) in the selected subsets. This is in contrast to the file-level results, in which the complexity metrics (e.g., AvgMaxNesting, CountPath) are the majority. In addition to this, the selected function-level metrics are quite different from the file-level metrics.

This exposes the limitation of previous research that focus only at a single level (function or file) for detecting the vulnerabilities (Shin and Williams [2008]; Chowdhury and Zulkernine [2011]; Alves et al. [2016b]). Given these results, we believe that for identifying the quality of software in terms of security, the function, file and project level metrics are complementary to each other.

To **validate the results above**, we used the same approach as for the file-level metrics. Tables 3.17 and 3.18 present the comparison between the Accuracy and Kappa of the classifier model built using the metrics selected by the genetic algorithm and the other 5 cases mentioned above.

Table 3.17: Comparison between the accuracy of the selected function-level metrics of Apache httpd, Xen HV and Glibc with several subsets.

	Apache httpd			Xen HV			Glibc		
	Metrics	**Accuracy**	**Kappa**	**Metrics**	**Accuracy**	**Kappa**	**Metrics**	**Accuracy**	**Kappa**
1 Top Correlated Metric	CountLineCodeExe	0,9438	0,0960	AltCountLineBlank	0,8434	0,0271	AltCountLineBlank	0,9840	0,0000
10 Top Correlated Metrics	CountLineCodeExe CountLineCode CountOutput CyclomaticModified AltCountLineCode CountLine CountSemicolon CyclomaticStrict CountStmtExe CountStmt	0,9719	0,2166	AltCountLineBlank CountOutput CountLineCodeExe AltCountLineCode CountLine CountLineCode MaxNesting CountLineBlank AltCountLineComment CountLineComment	0,8233	0,2152	AltCountLineBlank Essential CountPath CountLineCodeDecl CountOutput CountLineCodeExe CountLineCode CountLineBlank CountLine AltCountLineCode	0,9880	0,3961
Metrics Selected in all projects	CountLineCode CountStmtEmpty	0,9317	0,1569	CountLineCode CountStmtEmpty	0,8434	0,1455	CountLineCode CountStmtEmpty	0,9680	-0,0163
Metrics Selected in Most of Projects (3 and more)	CountLineComment CountStmt CountLineInactive CountLineCodeDecl CountLinePreprocessor CountSemicolon CountStmtDecl CountStmtExe CyclomaticModified Essential MaxEssentialKnots RatioCommentToCode	0,9639	-0,0072	CountLineComment CountStmt CountLineInactive CountLineCodeDecl CountLinePreprocessor CountSemicolon CountStmtDecl CountStmtExe CyclomaticModified Essential MaxEssentialKnots RatioCommentToCode	0,8474	0,2263	CountLineComment CountStmt CountLineInactive CountLineCodeDecl CountLinePreprocessor CountSemicolon CountStmtDecl CountStmtExe CyclomaticModified Essential MaxEssentialKnots RatioCommentToCode	0,9880	0,3961
All metrics	-	0,9679	0,3187	-	0,8594	0,2973	-	0,9880	0,3961
Selected Metrics by GA	-	0,9730	0,3703	-	0,8700	0,3530	-	0,9850	0,0990

Interestingly, we face the same situation as before: in all projects, except for Glibc, the classifier model is more accurate in the case where the selected metrics

Table 3.18: Comparison between the accuracy of the selected function-level metrics of Mozilla Firefox and Linux Kernel with several subsets.

	Mozilla Firefox			Linux Kernel		
	Metrics	**Accuracy**	**Kappa**	**Metrics**	**Accuracy**	**Kappa**
1 Top Correlated Metric	AltCountLineBlank	0,6759	0,3517	AltCountLineBlank	0,7296	0,1847
10 Top Correlated Metrics	AltCountLineBlank CountOutput CountLineBlank MaxNesting CountLine CountStmtDecl CountLineCodeDecl CountLineCode CountLineComment AltCountLineCode	0,7215	0,4430	AltCountLineBlank Essential CountOutput CountLine CountStmt CountLineCode AltCountLineCode CountStmtExe CountLineBlank CountLineCodeExe	0,7258	0,2677
Metrics Selected in all projects	CountLineCode CountStmtEmpty	0,6778	0,3555	CountLineCode CountStmtEmpty	0,7116	0,1653
Metrics Selected in Most of Projects (3 and more)	CountLineComment CountStmt CountLineInactive CountLineCodeDecl CountLinePreprocessor CountSemicolon CountStmtDecl CountStmtExe CyclomaticModified Essential MaxEssentialKnots RatioCommentToCode	0,6987	0,3973	CountLineComment CountStmt CountLineInactive CountLineCodeDecl CountLinePreprocessor CountSemicolon CountStmtDecl CountStmtExe CyclomaticModified Essential MaxEssentialKnots RatioCommentToCode	0,7135	0,2356
All metrics	-	0,7215	0,4430	-	0,7334	0,2710
Selected Metrics by GA	-	0,7432	0,4865	-	0,7570	0,3323

are used. This repetition of the same observation proves again that the genetic algorithm parameters were not properly set for the Glibc project.

The results of the cross-validation, presented in Table 3.19, again confirm that the metrics selected by the genetic algorithm for each project, except for Glibc, are the (near) best metrics for building the classifier.

Table 3.19: Cross-validation of the function-level metrics.

		Apache httpd		Xen HV		Glibc		Mozilla Firefox		Linux Kernel	
Metrics of	**Apache httpd**	0,973	0,3703	0,8635	0,3078	0,9840	0,4919	0,7148	0,4297	0,7239	0,2564
	Xen HV	0,9639	-0,0072	0,8700	0,3530	0,9880	0,5655	0,7196	0,4392	0,7220	0,2399
	Glibc	0,9719	0,3922	0,8474	0,2476	0,9850	0,0990	0,7015	0,4030	0,7249	0,2581
	Mozilla Firefox	0,9679	0,0000	0,8594	0,2772	0,9880	0,5655	0,7432	0,4865	0,7201	0,2445
	Linux Kernel	0,9719	0,2166	0,8514	0,2572	0,9920	0,6631	0,7186	0,4373	0,7570	0,3323

3.3 Software Metrics and Machine Learning to Detect Vulnerabilities

We now present our comprehensive experiment to study how effective software metrics combined with machine learning algorithms, are in distinguishing vulnerable from non-vulnerable code units. In practice, the goal is to contribute to answer the following Research Questions (RQs):

- **RQ1**. Can software metrics effectively be used to distinguish vulnerable code units from the non-vulnerable ones in different application scenarios?
- **RQ2**. How do different machine learning algorithms perform in this context?
- **RQ3**. Can the results of this experiment be generalized and applied to different types of software systems?

We aim to understand how the information provided by software metrics can be best used by machine learning algorithms to identify vulnerable code units (files, functions) with high levels of confidence within different circumstances, including different application scenarios that encompass diverse security concerns. To this end, this study considers:

1. Five representative software projects (Mozilla Firefox, Linux Kernel, Apache HTTPd, Xen and Glibc), used both individually and in combination;
2. Five combinations of software metrics of different types (complexity, volume, coupling and cohesion) collected at different levels of code (file and function). These different combinations of software metrics were selected based on dimension reduction approach presented in Section 3.2.2;
3. Five widely-used machine learning algorithms (Random Forest, Extreme Boosting, Decision Tree, SVM Linear and SVM Radial), considering different configurations to achieve the best prediction results;
4. Four application scenarios with diverse concerns regarding security (highly-critical, critical, low-critical, non-critical), which in practice are addressed by using different evaluation criteria (Recall, Informedness, F-Measure, Markedness). For instance, in the highly-critical systems scenario, detection and elimination of vulnerabilities is of high priority even if some false alarms are reported, therefore, a criterion that measures the ratio of detected vulnerable code units independently from false alarms seems to be of interest. In contrast, in the non-critical systems, the number of false alarms can be the main concern due to limited development resources, thus, a criterion that in addition to the correctly classified vulnerable code, strongly rewards low false alarms seems to be adequate.

The experimental process is divided in two phases, as shown in Figure 3.6. The first phase, **Preliminary Analysis**, is focused on the configuration and setting of the experiments. These are related to: *i)* selection of machine learning

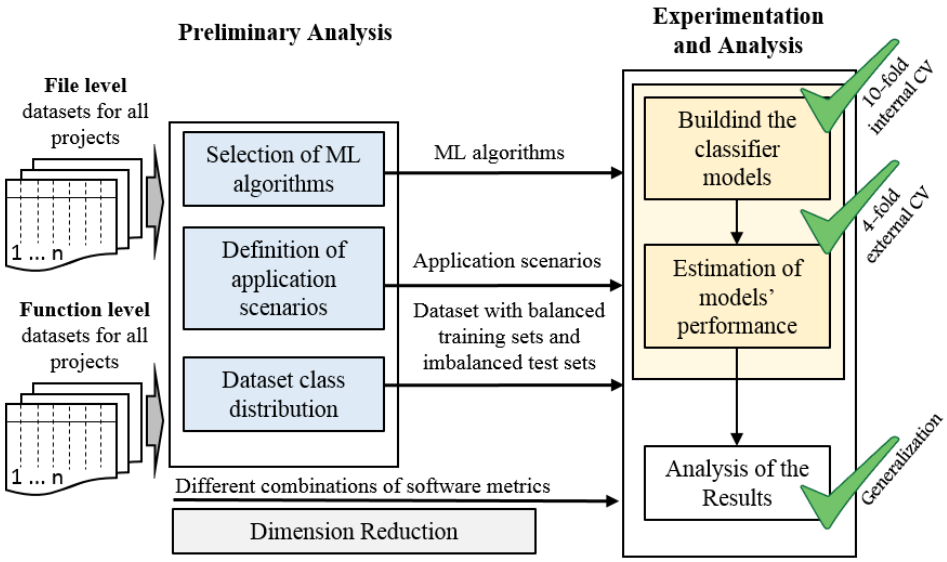


Figure 3.6: Methodology used in this work.

algorithms; *ii)* definition of application scenarios and selection of appropriate evaluation criteria for the scenarios; and *iii)* adjustment of the dataset class distribution.

The second phase, **Experimentation and Analysis**, is focused on running the experiments based on the configurations defined in the previous phase, and analyzing the results obtained. In practice, these experiments involve building and evaluating classification models using different machine learning algorithms, different combinations of software metrics, and diverse software projects within different application scenarios. In addition to and integrated with the above principal phases, we validate the approach and methods used as well as the results obtained and demonstrate whether the results can be generalized. These **Validation and Generalization (V&G)** activities are shown in the Fig. 3.6 with green check marks.

In the following sections, we introduce the Machine Learning algorithms selected, define the application scenarios and the decision criteria, and present the class distribution in the dataset. Then, we move to the build and evaluation of the classification models and the analysis of the classification results.

3.3.1 Machine Learning Algorithms

We selected several commonly used and recommended Machine Learning algorithms for detecting vulnerable code units based on software metrics. By referring to Ghaffarian and Shahriari [2017] and Alves et al. [2016a], that survey prediction models used for detecting vulnerabilities, the ones that seem to be the most commonly used in this area are: Decision Tree (S. and et. al. [1991]), Random Forest (Breiman [2001]), Support Vector Machine (Awad and Khanna [2015]) and (Boser et al. [1992]), and Logistic Regression (LR) (Dreiseitl and Ohno-Machado [2002]). These selected Machine Learning algorithms are presented and described in Section 2.4.

Since, in practice, LR and SVM with linear kernel usually present similar results (Westreich et al. [2010]), we use linear SVM in addition to radial SVM and discard LR. In addition to these, we also include the Extreme Gradient Boosted (Xboost) (Chen and Guestrin [2016]), as its good performance has been shown in many cases (Georganos and et al. [2018]).

The selected algorithms are used to perform supervised machine learning. Supervised classification requires that the data is totally labeled, as is the case in our work. The algorithms are tuned to achieve the best prediction result at the cost of having longer training time. In the case of Xboost, Linear and Radial SVM, a list of values (based on literature) are given to the algorithms for each parameter to try different combinations and the best result is selected in each case. In the case of Random Forest and Decision Tree, the recommended default values from the literature are used for each parameter.

3.3.2 Application Scenarios and Decision Criteria

To improve the effectiveness of the Machine Learning models, it is important to adequate the evaluation criteria to the relevant application contexts. We consider four distinct scenarios where security assurance has different levels of relevance, depending on the criticality level of the applications being developed and also on the availability of resources to deal with security problems. The four scenarios analyzed were adapted from Nunes et al. [2018], where the authors define different real-world scenarios of applications to benchmark static analysis tools. We analyzed the specific characteristics of each scenario and selected an appropriate criterion associated to each one in order to evaluate the classifiers built on top of the selected software metrics. The scenarios and associated criteria are:

- **Highly-Critical**: this scenario represents highly business or safety critical systems with demanding security requirements (e.g., e-banking and e-health), in which the detection and elimination of security vulnerabilities is of high priority (because a successful security attack may cause serious damages to the system, to business, or to people's life). Thus, the classifier models should be able to detect the highest number of vulnerable code units, even if some false positives are reported. For this scenario, we choose **Recall** as criterion to evaluate the classifiers, as it measures the ratio of vulnerable code units that are correctly classified independently from false positives.
- **Critical**: this scenario represents not highly but still critical systems (e.g., e-commerce web applications and large scale social networks) in which an exploited vulnerability usually reflects sensitive data breaches or considerable financial losses. In such scenario, classifiers should detect the highest number of vulnerabilities while avoiding reporting too many false positives as the resources available to fix and remove vulnerabilities need to be used appropriately. For this reason, we chose **Bookmaker Informedness** as criterion, as it still gives a high importance to true positive rate while moderately penalizing classification models with high false positive rates.

- **Low-Critical**: this scenario includes systems that are less critical and less exposed to attacks. Projects developing these systems usually have limited budget to be allocated for finding and fixing vulnerabilities. Thus, both detecting and eliminating the highest number of vulnerabilities and spending less resources for analysing false positives have equal priority. In this scenario, **F-Measure** that evenly combines precision and recall, is an appropriate criterion.
- **Non-Critical**: this scenario includes non-critical systems from a security perspective (i.e., systems that are not usually exposed to attackers). Thus, we are more concerned with the number of false alarms due to tight budget and resource restrictions, although we still want to detect vulnerable code and eliminate vulnerabilities. **Markedness** is an appropriate criterion in this context, as it rewards low false alarms and at the same time does not ignore true positives.

It is important to mention that, bookmaker Informedness and Markedness are unbiased metrics that characterize the effectiveness of the predictors considering the proportion of the classifications, similarly to what is done with betting odds (Powers [2020]). Therefore, Bookmaker Informedness characterizes the effectiveness of the predictor considering measures of the proportion of outcomes, and the Markedness quantifies how marked a condition is for the specified predictor, versus chance.

Table 3.20: Summary of the application scenarios and their corresponding criteria (Antunes and Vieira [2015]).

Scenario	Criterion	Formula	Definition
Highly-Critical	Recall	$\frac{TP}{P} = \frac{TP}{TP+FN}$	Represents the ratio of vulnerable code units that are correctly classified as vulnerable.
Critical	Bookmaker Informedness	$\frac{TP}{P} - \frac{FP}{N} = \frac{TP}{TP+FN} - \frac{FP}{TN+FP}$	Combines TP and FP rates but still gives a high importance to the number of vulnerable units that are correctly classified and moderately penalizes classification models with high number of FP.
Low-Critical	F-Measure	$2 * \frac{precision * recall}{precision + recal} = \frac{2 * TP}{2 * TP + FN + FP}$	Represents the harmonic mean of Recall and Precision, thus evenly combines TP and FP rates.
Non-critical	Markedness	Precision + Inverse Precision $- 1$ $= \frac{TP}{TP+FP} + \frac{TN}{FN+TN} - 1$	Quantifies how consistently the outcome has the classifier as a marker. It does consider both true and false alarms, but in practice, it rewards the low false positive rate.

More details about the selected criteria are presented in Table 3.20. In the formulas, True Positive (TP) represents the number of vulnerable code that are correctly classified, True Negative (TN) represents the number of non-vulnerable code that are correctly classified, False Positive (FP) represents the number of non-vulnerable code units that are misclassified as vulnerable and False Negative (FN) represents the number of vulnerable code that are misclassified as non-vulnerable.

3.3.3 Class Distribution in the Dataset

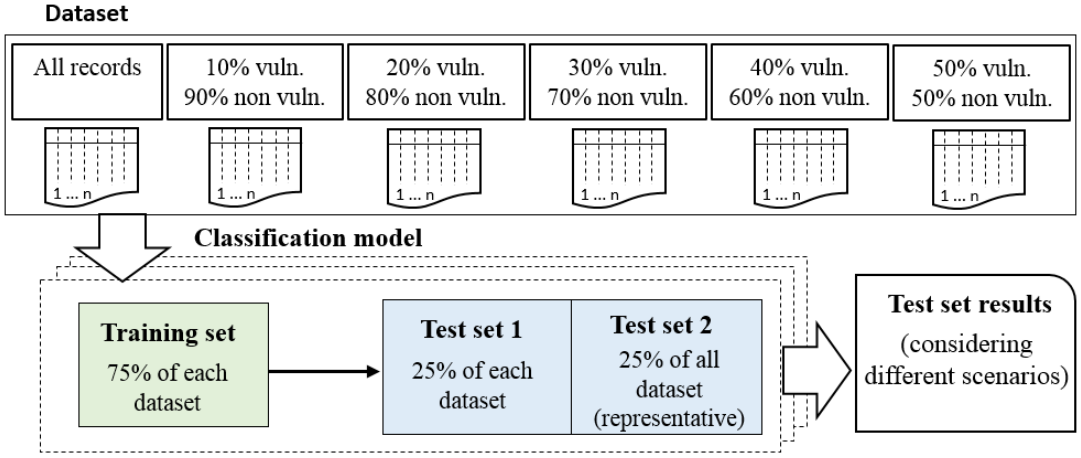
Table 3.21 presents projects information regarding the number of files and functions used in this experiment. As can be observed, the dataset is different from the one described in Table 3.3. In fact, the previous dataset contained only one record to each file and each function and the current dataset contains several records for each file and function. Therefore, as we have information regarding each commit of each project, we decided to include all the records (representing the evolution of the software metrics for each file and function over time).

Table 3.21: Summary of the dataset.

Software Projects	# Files			# Functions		
	Total	# Vulnerable	% Vulnerable	Total	# Vulnerable	% Vulnerable
Mozilla Firefox	185994	3379	1,82%	1418482	2780	0,20%
Linux Kernel	383622	8712	2,27%	1910776	3021	0,16%
Apache httpd	3031	94	3,10%	17046	50	0,29%
Xen HV	6196	278	4,49%	35430	241	0,68%
Glibc	21843	94	0,43%	23790	24	0,10%

The dataset is quite imbalanced, as the vulnerable code units make a small fraction of the whole dataset (e.g., 2.27% in the case of Linux Kernel files). In such cases, research shows that machine learning algorithms tend to be overwhelmed by the large class and ignore the small ones (Chawla et al. [2004]). On the other side, transforming a representative dataset into a balanced dataset (either by undersampling or by oversampling) may cause the loss of information about the frequency of each class and, thus, affecting the accuracy of the classification models (Batista et al. [2004]). For this reason, we performed an analysis to find out how balanced the dataset should be in order to build high performance classifiers for vulnerability detection (i.e., models with high true positive and low false positive rate). The process used is described in Figure 3.7.

Figure 3.7: Undersample process.



In practice, we apply one of the most effective (in terms of performance) and efficient (in term of time) strategies to deal with imbalanced data, which is to moderately undersample the majority class (Estabrooks et al. [2004]), to gradually balance the dataset (from a fully representative and imbalanced dataset to a 100% balanced dataset) and observe the impact on the performance. This allows to select a dataset with the near best class distribution that results in the near best performance compared to others.

In all experiments, 75% of the resampled dataset was used to train the machine learning algorithms and 25% of it (disjoint from the training sets) was used to test them (named as **TS1**). In addition, to guarantee a fair and representative evaluation of the classification models, we (randomly) created an additional test set composed of 25% of the whole dataset (**TS2**), which is fully imbalanced and is ensured to be disjoint from the training sets. By doing this, we aim to understand how the estimation made by a balanced test set differs from the estimation made by an imbalanced, but representative test set.

Table 3.22 presents the characteristics of the resampled datasets for the Linux Kernel project. As we can observe, we moved from a fully imbalanced dataset composed by 2.27% of vulnerable files and 97.73% of non-vulnerable files, to a balanced dataset with 50% of vulnerable files and 50% of non-vulnerable files. Also, the number of files used to test the machine learning algorithms are presented in *Test set 1* and *Test set 2* columns. Linux Kernel was chosen due to the fact that it has a higher number of reported vulnerabilities than other projects, so a low number of vulnerable records would not be a threat to the validity of the results.

Table 3.22: Resampled datasets (Linux Kernel files).

	Vulnerable Files	Non Vulnerable Files	Total	Training set	Test set 1 (TS1)	Test set 2 (TS2)
Imbalanced	8712 (2,27%)	374910 (97,73%)	383622	287717	95905	95905
	8712 (10%)	78408 (90%)	87120	65340	21780	95905
	8712 (20%)	34848 (80%)	43560	32670	10890	95905
	8712 (30%)	20328 (70%)	29040	21780	7260	95905
	8712 (40%)	13068 (60%)	21780	16335	5445	95905
Balanced	8712 (50%)	8712 (50%)	17424	13068	4356	95905

Figure 3.8 (x-axis: % of vulnerable records in the dataset (from 2.27% to 50%), y-axis: true positive rate (left) and false positive rate (right)) shows how performance, in terms of true positive rate and false positive rate estimated using resampled test set (TS1), changes when the training set becomes more balanced. For all the machine learning algorithms considered, we observe that the true positive rate increases (e.g., from 0.54 to 0.92 in the case of Random Forest and from 0.08 to 0.73 in the case of Decision Tree). This means that more vulnerable code units are detected and less vulnerable code units are misclassified as non-vulnerable. Thus, for highly critical systems where one wants to detect as many vulnerabilities as possible (regardless of the false alarms), it is quite effective to

balance the dataset when the number of vulnerable records is lower than the number of non-vulnerable ones.

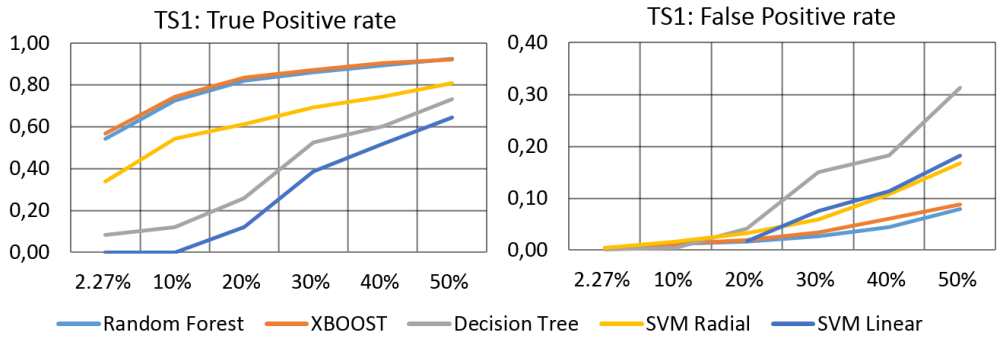


Figure 3.8: Impact of undersampling on performance.

Another observation is that, the false positive rate increases for all algorithms (e.g., from 0.003 to 0.08 in the case of Random Forest and from 0.0007 to 0.31 in the case of Decision Tree), which means that a higher number of non-vulnerable code units are misclassified as vulnerable. Thus, for scenarios in which there are limited resources for fixing or removing vulnerabilities, undersampling the non-vulnerable class to balance the dataset does not seem to be a good approach. Similar results are obtained for true positive rate and false positive rate, using the imbalanced test set (TS2).

As a result, since we are more concerned about detecting vulnerable code units and aim to improve the tools and techniques in this regard, **we have decided to use the totally balanced (50% vulnerable code units) datasets for training the machine learning algorithms** (in experimentation and analysis phase).

We also conducted a more detailed comparison between the classifiers using balanced and imbalanced *test sets*. For this comparison, we used all machine learning algorithms, trained using a totally balanced training set and tested using both balanced (TS1) and imbalanced (TS2) test sets, and evaluated the classification models by using the four criteria representing the four scenarios under study.

As shown in Figure 3.9, the Recall and Informedness obtained using TS1 are in par with the results obtained using TS2. This means that using either a balanced or an imbalanced test set does not influence the classification results when highly-critical and critical scenarios are the target of the analysis. But we observe lower values for F-measure and Markedness in TS2 across all machine learning algorithms. This is caused by the high number of false positives compared to true positives, which comes naturally from the TS2 that has a much higher percentage of non-vulnerable records. Thus, **we have decided to use imbalanced but representative test sets in the following experiments, in order to have realistic performance estimation for all scenarios.**

Due to the time required to carry out all of these experiments (included in the preliminary analysis), we decided to perform them only at file level. Because

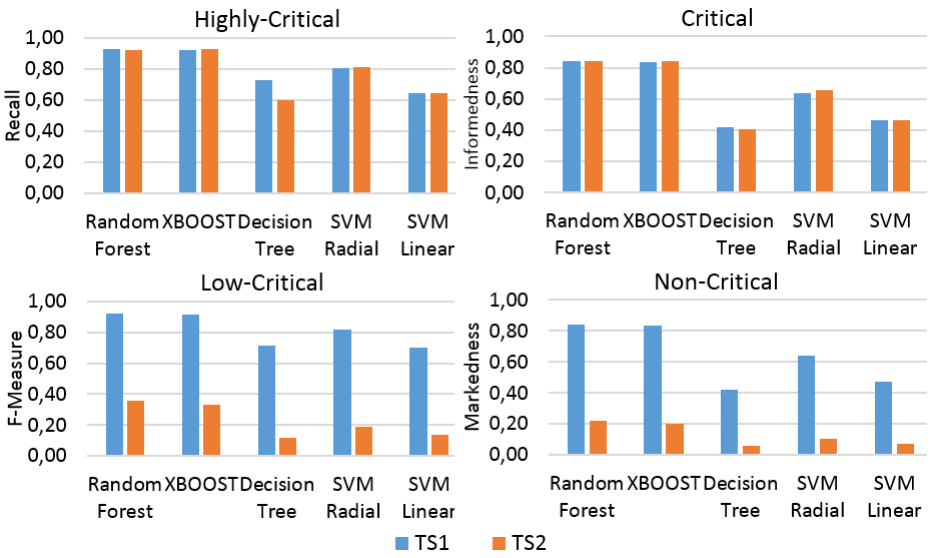


Figure 3.9: Balanced versus imbalanced representative test sets.

of this, our decision regarding how to train and test the models may not perfectly fit in function-based experiments, but we believe that the implications are negligible, due to the fact that the nature and context of the problem is quite similar.

3.3.4 Experimentation and Analysis

The second phase of the study consists of running the **experiments**. As shown in Figure 3.10, the data (balanced training sets and representative test sets belonging to all projects at both file and function levels) are prepared according to the configurations determined in the previous phase and then passed to the selected Machine Learning algorithms. The classification models are built over the dataset of the five different projects at file and function levels by considering the several combinations of software metrics and the different application scenarios. It is worth repeating that the machine learning algorithms are trained using *balanced training sets* and tested using a *representative test set* in order to build more accurate vulnerable code detectors and have more realistic performance estimations. Internal and external cross-validation (CV) is performed in all cases, as discussed next.

V&G - Internal Cross Validation: In order to avoid any overfitting that might be caused by unrepresentative training sets, internal cross validation is necessary. Cross-validation is a statistical resampling technique used to estimate the performance of machine learning models (Yu [2002]). Using this technique, data is split into k subsets or folds of equal size. Each time, one fold is used as test set and the remaining k-1 folds are used to train and fit the model. In this work, we use an **internal 10 fold cross validation** for building the models, helping to achieve a fair estimation of the performance for each individual model.

V&G - External Cross Validation: Internal cross validation might not be enough to ensure fair comparison between distinct models due to the fact that the initial training and test sets might not be representative of the whole dataset.

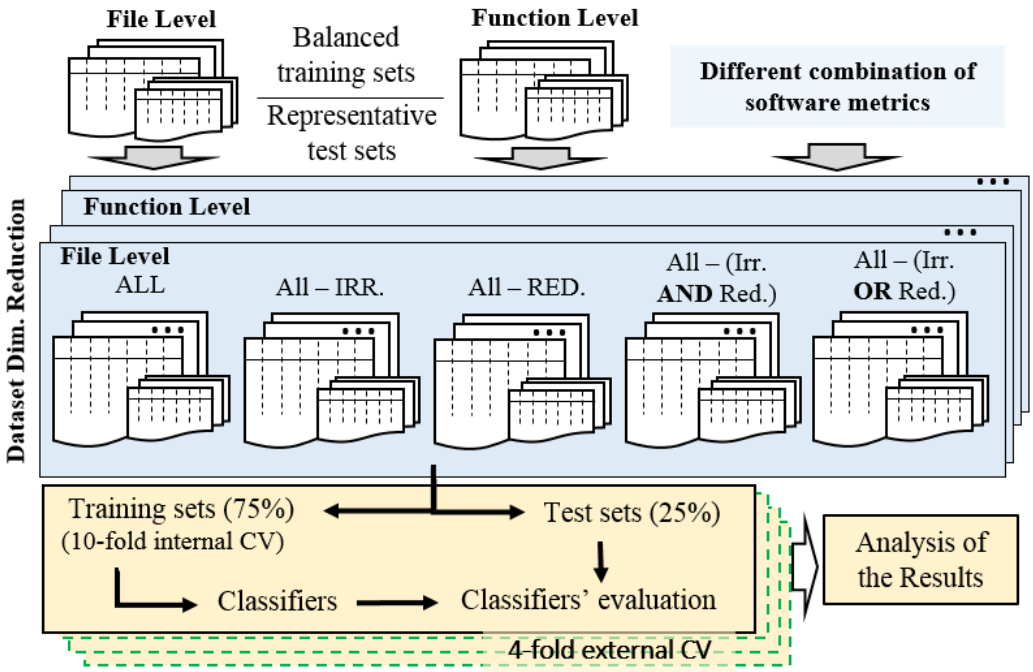


Figure 3.10: Process of experimentation and analysis.

For this reason, in this work, we use an **external 4-fold cross validation** to validate the classification models built. In practice, we divide the whole dataset into 4 folds. Each machine learning algorithm is executed four times; each time, it uses one fold for testing and 3 folds for training (which internally uses a 10 folds cross validation). The final performance estimation of each classification model is an average of the four estimations.

V&G - Generalization Assessment: We conducted two sets of tests to understand to which extent we can generalize the obtained results:

1. The first set of tests is focused on inter-project cross assessment. In these tests, the machine learning algorithms are trained using the dataset of one project (e.g., Linux Kernel) and are tested using the dataset of the other projects (e.g., Mozilla Firefox). This helps to understand how machine learning algorithms perform in new situation.
2. In the second set of tests, the machine learning algorithms are trained using a combined dataset including all projects and tested using the dataset of each project individually. This helps understanding whether it is helpful to combine all existing information from source code of different software projects to achieve a better result.

In the following sections, we present and analyze the results obtained during the experimentation and analysis phase, including the performance of the machine learning algorithms and the generalization of the approach.

3.3.4.1 Performance of the Machine Learning Algorithms

We first focus on the results obtained for each project individually and then make a comparison. Figures 3.11 and 3.12 present the results obtained respectively

for file and function level metrics of the Linux Kernel project. Both include the results obtained by all Machine Learning algorithms for different scenarios over five combinations of software metrics (dimension reduction presented in Section 3.2.2). It is worth reminding that all 5 software projects are analyzed by using four criteria representing four different scenarios. In fact, the assumptions regarding the criticality level of the projects are made based on the scenarios.

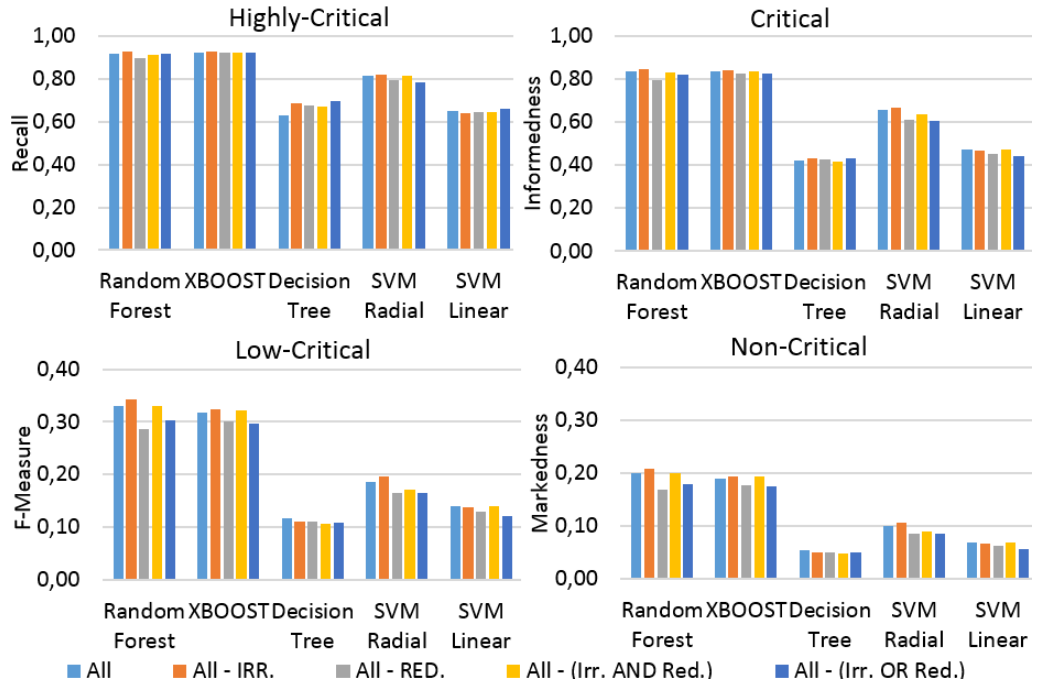


Figure 3.11: File level results for Linux Kernel project.

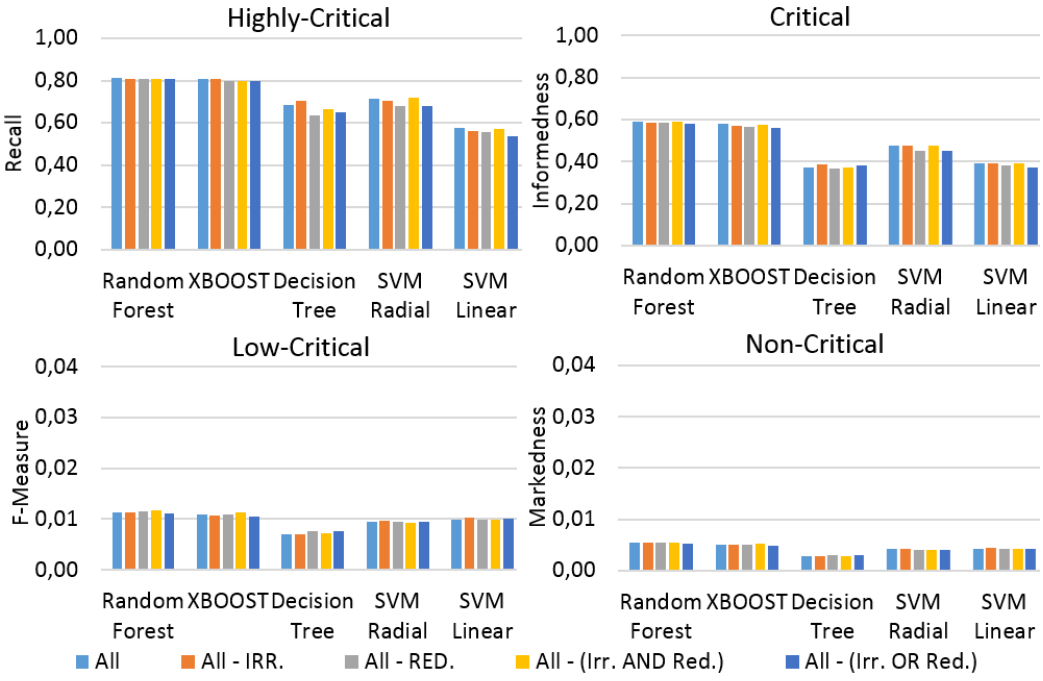


Figure 3.12: Function level results for Linux Kernel project.

File level results show that the best performance is always achieved by Random Forest and Xboost algorithms. As expected from the non-linear nature of the dataset, radial SVM always achieve a better performance than linear SVM, which is almost in par with Decision Tree.

Among different combinations of software metrics, the combination from which the irrelevant metrics are eliminated *slightly* shows a better result than other combinations in most cases. In Figure 3.11, we can also see that the combination in which the redundant metrics are eliminated shows (slightly) worse results than the combination with all metrics. In contrast, function level results show no significant difference between these combinations. This happens because the function-level dataset is not considered as a high-dimensional dataset (it only has 28 features), and in such cases it is hard to achieve a better result with dimension reduction. However, this is not always the same for other projects. See the results of file level metrics and function level metrics for Glibc project in figures 3.13 and 3.14, respectively.

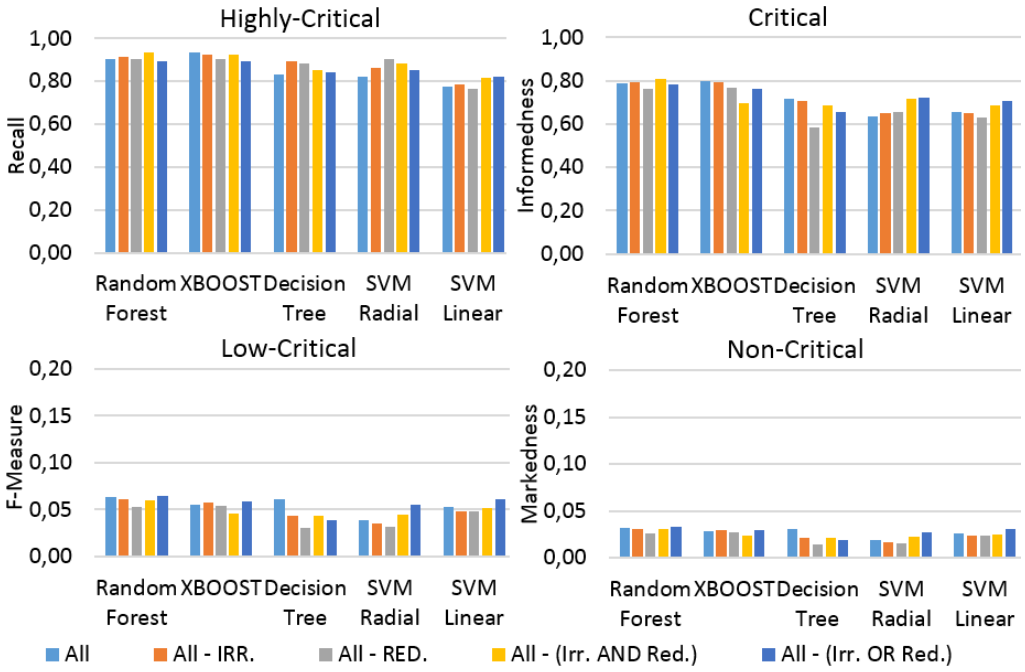


Figure 3.13: File level results for Glibc project.

After analysing the results of all projects and all algorithms, we can state that, **dimension reduction, does not always help to achieve a better performance**. In fact, dimension reduction has to be done carefully and several techniques should be tried depending on the classification model in use and the characteristics of the dataset in order to achieve a better performance.

Regarding the effectiveness of using software metrics and Machine Learning to detect vulnerable functions, we can conclude that, although the machine learning algorithms could achieve a reasonable performance in terms of Recall and Informedness (highly critical and critical scenarios), the results for F-measure and Markedness (low-critical and non-critical scenarios), which are highly dependent on the number of false positives compared to true positives (refer to

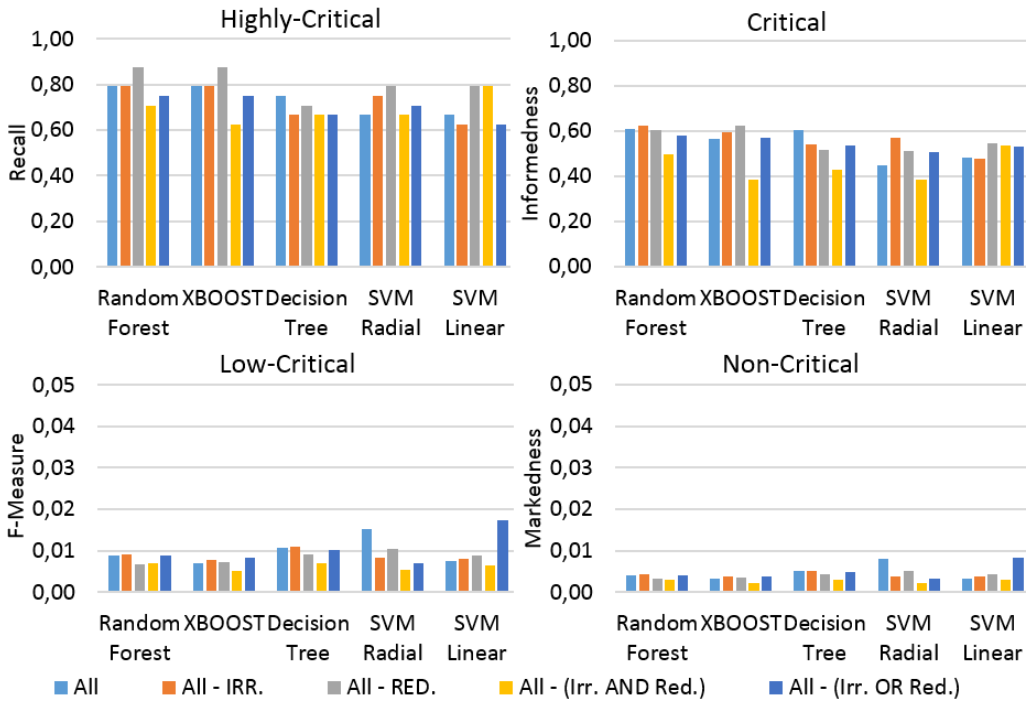


Figure 3.14: Function level results for Glibc project.

Table 3.20), are not convincing at all. Despite having high true positive (TPR = TP/P) and low false positive rates (FPR = FP/N), having a very imbalanced test set leads to a high number of false positive cases when compared to the number of true positive cases.

Similar observations can be pointed for the **function level results** presented in Figure 3.12, with the difference that the performance of the algorithms using file level metrics is usually higher than when using function level metrics. Also, the difference between machine learning algorithms is more visible in file level results. For example, we cannot see any difference between the algorithms in terms of F-Measure and Markedness in the figure. This happens due to the fact that the function-level data is even more imbalanced than the file-level data (refer to Table 3.21).

To make a comparison between different projects, we present in Figure 3.15 the results obtained for all projects over the data sets with all file level metrics. In general, the results for the different projects are quite different mainly due to the fact that the characteristics of the datasets (i.e., size and distribution of classes) are different for each project. In most cases, the best performance is achieved for the Linux Kernel dataset, which is the biggest project and has more vulnerable code units. This means that the machine learning algorithms had more evidences and more balanced information to avoid overfitting and learn (of course not equally) about both classes involved in the dataset. We also have high Recall and Informedness for Glibc, but, by looking to the very low F-measure and Markedness values, we can conclude that the high true positive rate in this case is achieved thanks to highly overfitted models.

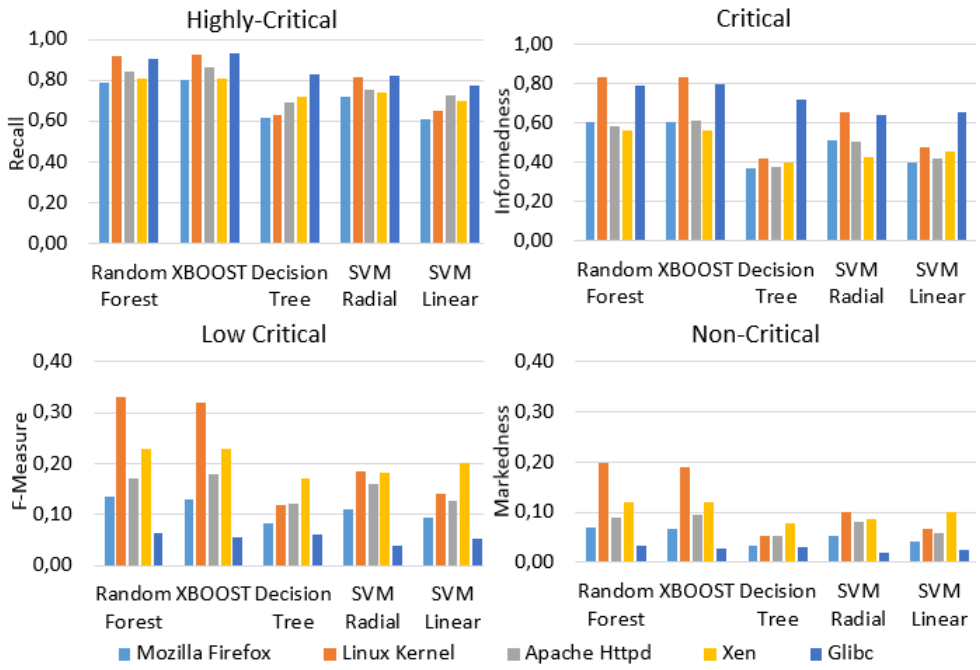


Figure 3.15: File-level results for all projects over all software metrics.

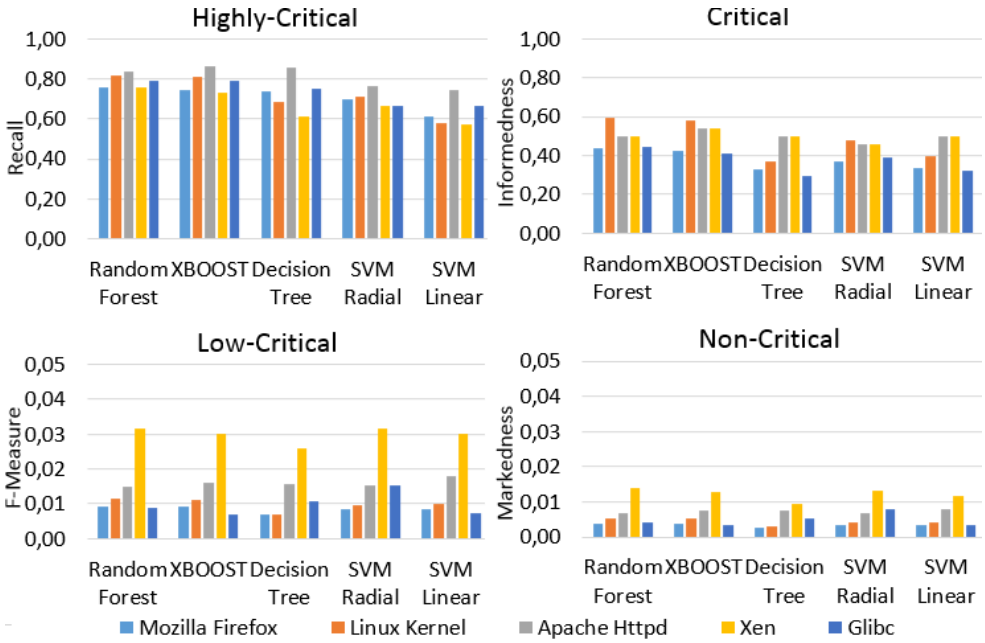


Figure 3.16: Function-level results for all projects over all software metrics.

Interestingly, the results achieved by the two ensemble algorithms, Random Forest and Xboost, are quite similar in the case of all projects, for both file and function level metrics (see figures 3.15 and 3.16). Random Forest and Xboost are tree-based algorithms and, in both cases, the performance of the model depends on two distinct sources of error: bias and variance. Gradient boosting models deal with these sources of error by boosting for many rounds at a low learning rate. In contrast, Random Forest models deal with them via the number of trees and tree depth. Achieving very similar results by these algorithms in almost all

cases may imply that both models were able to achieve their best model with our dataset and no bias or variant could be reduced by neither methods due to the limitations that exist in the dataset (e.g., being imbalanced with imperfect labeling).

3.3.4.2 Generalization Assessment

To understand to which extent we can generalize the results and how the Machine Learning algorithms perform in new (previously unseen) situations, we performed a inter-project cross assessment, where data of a specific project are used for training and data of the other projects are used for testing.

At the file level and using data of Linux Kernel as training set, we observe that the performance decreases in all projects, except in the case of the Linux Kernel itself, whose data is used for training the machine learning algorithms (see Figure 3.17). An interesting observation is that Linear SVM and DT seem to make better classifications than other machine learning algorithms when the test set is completely unknown to the classifiers. This suggests that these algorithms are able to build more generalizable models than others, thus being more suitable for unseen code. This is simply because, they build simpler models, which is more appropriate when the data is more non-parametric in nature (i.e., when we cannot make assumptions about the distribution of data).

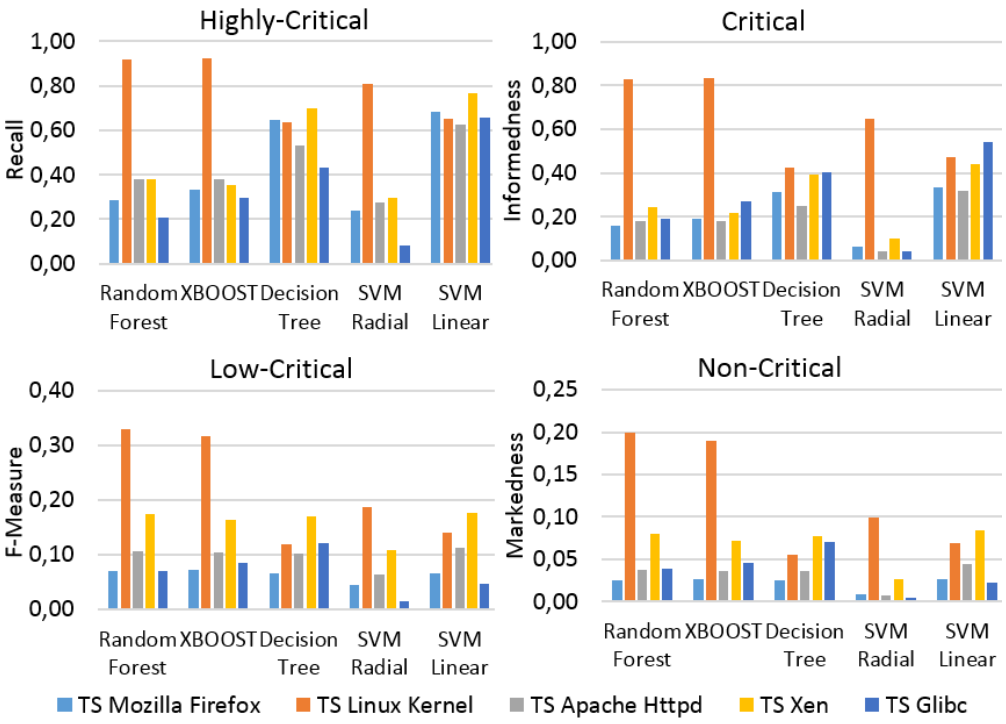


Figure 3.17: File-level inter-project cross-validation results (Linux Kernel data is used as training set).

At function level and using data of Linux Kernel as training set (see Figure 3.18), all classifiers seem to perform similarly. Interestingly, for Low Critical and Non-Critical scenarios, Xen achieves a better result than the other projects.

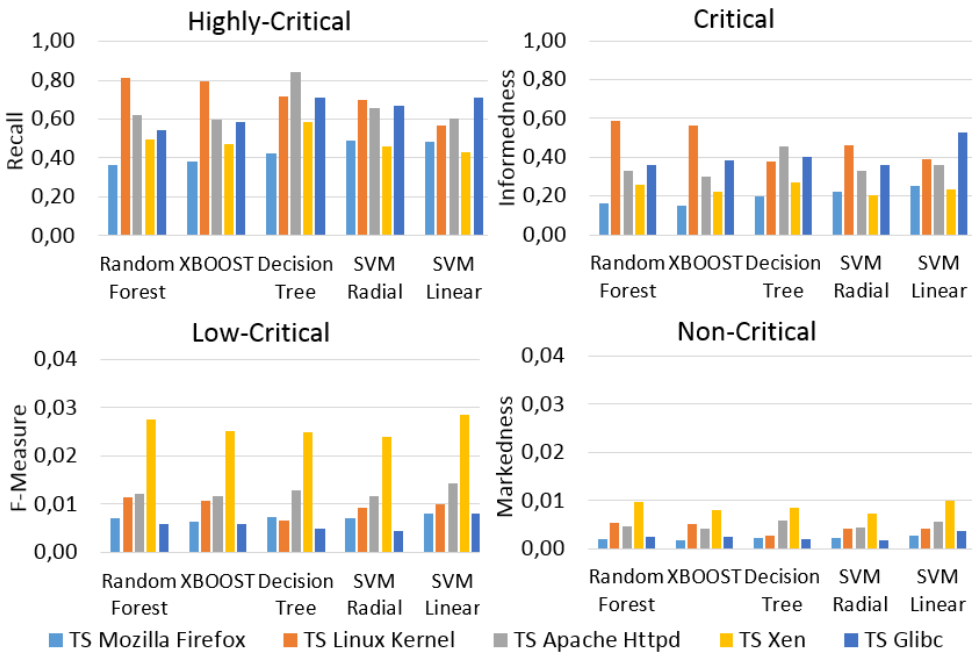


Figure 3.18: Function-level inter-project cross-validation results (Linux Kernel data is used as training set).

This happens due to the fact that this small project has a more balanced test set compared to other projects.

The same model with more balanced test set, gives less false positive alarms compared to the number of true positive cases, which leads to achieve higher F-Measure and Markedness. The same observations are seen when data of Mozilla Firefox is used as training set in both file and function levels, but in other cases, when the data of the small projects are used for training, we observed that the performance of the classifiers is way lower and all classifiers perform similarly in both file and function level. This happens because the training set is small and there is not enough variation in training set.

The results of the experiments in which the Machine Learning algorithms are run over the combined dataset, are presented in figures 3.19 and 3.20 for files and functions, respectively. We can observe that the performance of the classifiers is *slightly* degraded when we use a dataset composed of all 5 projects for building the classification models. This potentially means that classifiers are able to find similar characteristics and patterns in the code of five different projects, thus achieving a reasonable performance level.

The results are similar for function level metrics (figure 3.20). This is a promising observation as it may mean that we can build a dataset with higher diversity (including different types of software project), which is quite helpful for vulnerebility prediction of unseen code but still have a reasonable level of performance.

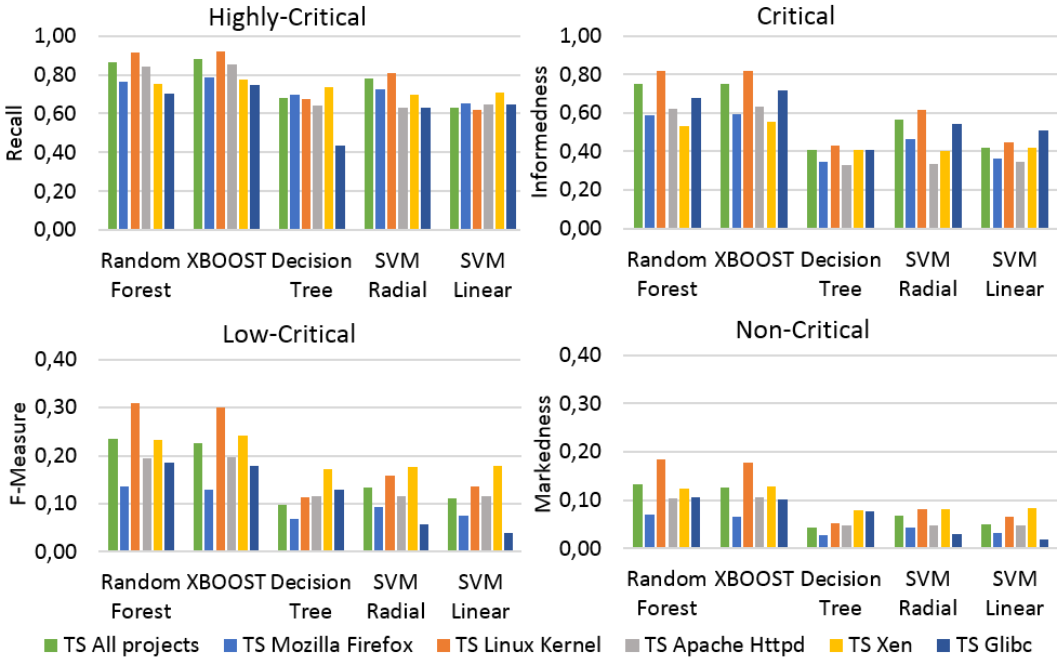


Figure 3.19: File-level generalization results.

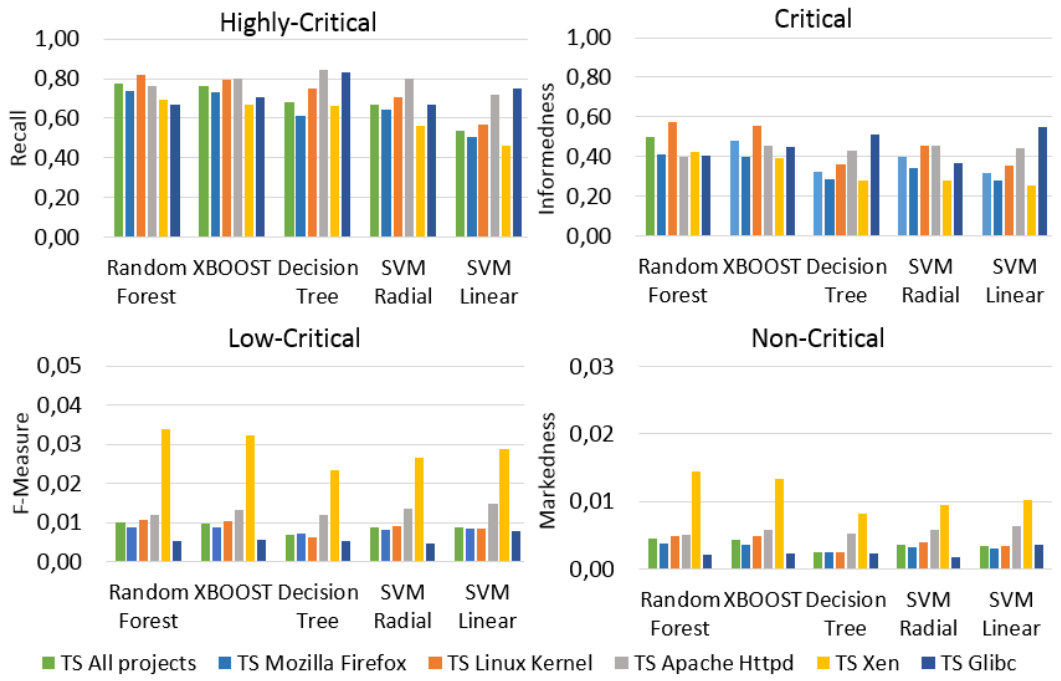


Figure 3.20: Function-level generalization results.

3.3.5 Analysis of the Classification Results

We conducted several more detailed analyses to better understand the results of the Machine Learning algorithms. As depicted in Figure 3.21, we followed three approaches:

1. **Intersection analysis** that aims at understanding the intersections of

the classification (false or true) results of different ML algorithms.

2. **Misclassified code analysis** that aims at understanding the characteristics of misclassified code units when compared to correctly classified code units.
3. **Code review** that aims at showing that the code units that are wrongly classified as vulnerable might indeed include unknown vulnerabilities (that can be the target of future attacks), and that the ones that are correctly classified as non-vulnerable are not necessarily flawless.

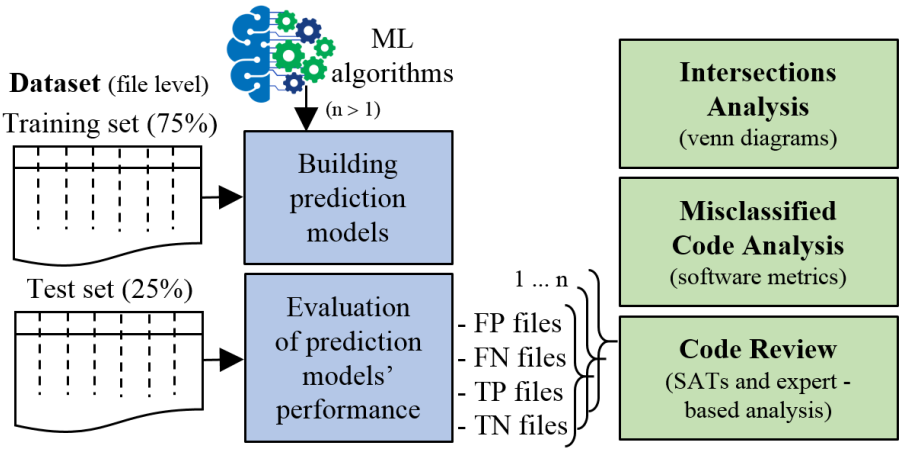


Figure 3.21: Profound analysis of prediction results.

We present the detailed results obtained for the **Linux Kernel** project at **file level** because it is the project with the higher number of known vulnerabilities (refer to Table 3.21), thus allowing to build a more balanced dataset with a sufficient number of vulnerable records to avoid overfitting, as much as possible. Nevertheless, we also conducted the *intersections analysis* and the *analysis of misclassified code* for the Mozilla Firefox project and similar results were observed.

3.3.5.1 Intersection Analysis

The results of the prediction models are summarized in Table 3.23. The total number of vulnerable records and total number of non-vulnerable records comprise only 25% of the entire dataset (test set). Thus, 2178 vulnerable and 93727 non-vulnerable files of the Linux Kernel project are considered in the analysis. Table 3.23 also presents the number of *vulnerable files* that were classified as vulnerable and as non-vulnerable (true positive and false negative classifications, respectively), and the *non-vulnerable files* that were classified as non-vulnerable and as vulnerable (true negative and false positive classifications, respectively).

It is possible to observe similar patterns between Linear Support Vector Machine and Decision Trees. However, showing a comparable performance does not imply that the code is classified or misclassified similarly by these classifiers. For this reason, we decided to analyse their behaviour in more detail. We use Venn

ML Algorithm	# Vulnerable		Total vuln.	# Non Vulnerable		Total non vuln.
	TP	FN		TN	FP	
RF	2018	160	2178	85712	8015	93727
Xboost	2032	146	2178	85319	8408	93727
DT	1357	821	2178	75257	18470	93727
SVM L	1396	782	2178	77437	16290	93727
SVM R	1793	385	2178	77935	15792	93727

Table 3.23: Prediction models' evaluation results (files ofLinux Kernel project).

diagrams to better understand the intersections of the classification results (false or true) of different machine learning algorithms. Figure 3.22 presents Venn diagrams showing all possible intersections between the subset of code units classified as vulnerable or non-vulnerable by the different ML algorithms.

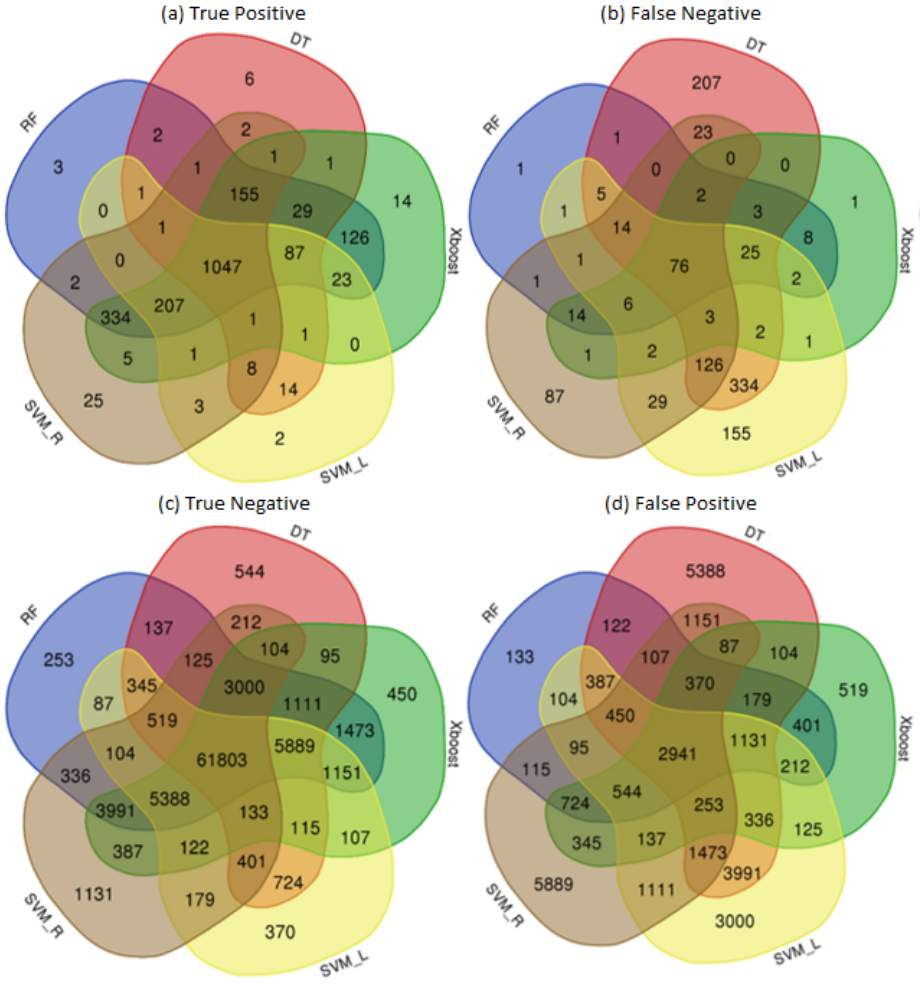


Figure 3.22: Venn Diagrams of the Classification Results of 5 ML Algorithms.

The diagrams show that 1047 vulnerable files (48%) are detected by all models (a), and that 76 vulnerable files (3.5%) are missed by all classifiers (b). It also shows that 61803 non-vulnerable files (i.e., no vulnerability was reported up to the date of the dataset creation) (66%) are classified as non-vulnerable (c), and that 2941 non-vulnerable files are classified as vulnerable (3.1%) by all classifiers

(d). These four groups of files classified in the same way by the five prediction models probably share characteristics that lead them to the same class, which will be discussed next.

3.3.5.2 Misclassified Code Analysis

To better understand the results above, we first compared the characteristics of the vulnerable/non-vulnerable files misclassified by all classifiers with the characteristics of the files that are correctly classified by all classifiers. All software metrics were analyzed and we observed that the results are similar across some software metrics that are highly correlated/redundant (e.g., CountLineCode and AltCountLineCode).

Figure 3.23 presents the average value of several non-redundant and representative software metrics (at least one metric from each type including volume, complexity, coupling and cohesion metrics) for each group of files. Interestingly, correctly classified vulnerable files (TPs) and wrongly classified non-vulnerable files (FPs) show very similar structural characteristics (i.e., huge and complex). This can be observed by their close average values.

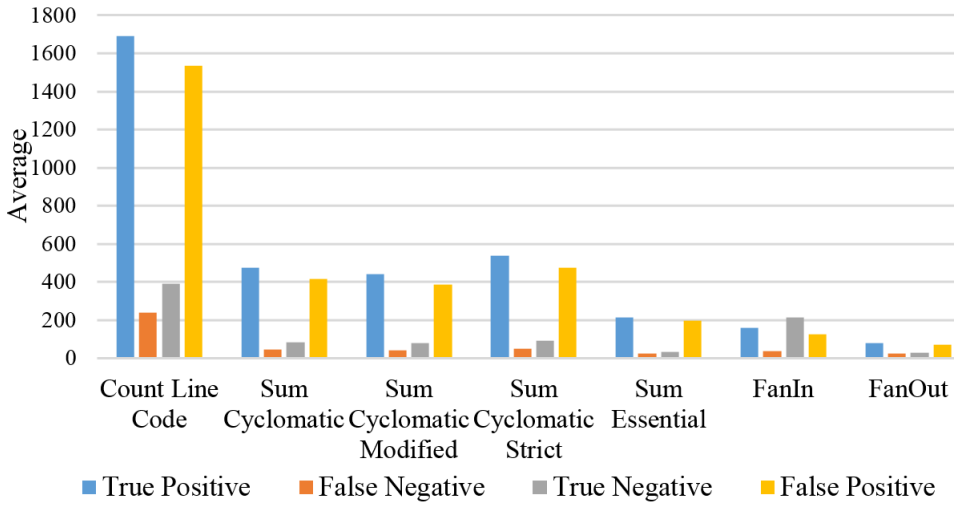


Figure 3.23: Comparing TPs, FNs, TNs, and FPs in terms of Software Metrics.

The same is true regarding the correctly classified non-vulnerable files (TNs) and the wrongly classified vulnerable files (FNs). There is, however, a considerable difference between these two groups (files classified as vulnerable and files classified as non-vulnerable by the five prediction models). The vulnerable files classified as non-vulnerable are small and simple in terms of structure. In contrast, most of the files and functions incorrectly classified as vulnerable are huge or complex. An example of a misclassified vulnerable file (from FNs) (false negative) from the Linux Kernel source code is presented below:

```
1 /*
2  * File Path: fs/ramfs/file-mmu.c
3  * Software Metrics values:
4  * CountLineCode: 15
5  * SumCyclomatic: 0
```

```
6  * SumCyclomaticMod: 0
7  * SumCyclomaticStrict: 0
8  * SumEssential: 0
9  * CountPath: 0
10 * FanIn: 0
11 * FanOut: 0   */
12
13 #include <linux/fs.h>
14 #include <linux/mm.h>
15 #include <linux/ramfs.h>
16 #include "internal.h"
17
18 const struct file_operations ramfs_file_operations = {
19   .read    = new_sync_read,
20   .read_iter  = generic_file_read_iter,
21   .write    = new_sync_write,
22   .write_iter = generic_file_write_iter,
23   .mmap    = generic_file_mmap,
24   .fsync    = noop_fsync,
25   .splice_read  = generic_file_splice_read,
26   .splice_write = generic_file_splice_write,
27   .llseek    = generic_file_llseek,
28 };
29 const struct inode_operations ramfs_file_inode_operations = {
30   .setattr  = simple_setattr,
31   .getattr  = simple_getattr, };
```

We added the first 11 lines just to provide some information about the file. The file path is presented in line 2, and the values of several representative software metrics are included in lines 3 to 11. Such values show how **simple** the file is. Indeed, it is impossible to indicate this file as vulnerable by looking only to software metrics, but we are aware of one exploitable vulnerability that has been reported for this file (i.e., *CWE-264 - Permissions, Privileges, and Access Controls*). The vulnerability consists of a **Wrong Assignment Value** (according to the Orthogonal Defect Classification (Chillarege and et. al. [1992])) in line 26, which allows local users to cause a denial of service (system crash). To fix this vulnerability the line should be simply replaced by *.splice_write = iter_file_splice_write,*. Just to offer some context, the Orthogonal Defect Classification (ODC) is a systematic framework for Software Defect Classification that uses semantic information from defects to extract cause-effect relationships in the development process. Thus, provide fast and effective feedback to developers (Chillarege et al. [1996]). The defect types used are: function, interface, checking, assignment, timing/serialisation, build/package/merge, documentation and algorithm (Lopes Margarido et al. [2011]).

As for the false positive cases, as mentioned before, the source of information regarding the vulnerabilities in the dataset is limited to known security reports. Consequently, files without reported vulnerabilities are not necessarily flawless. For this reason, given the above results, it is quite probable that some of the cases considered as false positives are indeed vulnerable, or at least untrustworthy (although no vulnerability has yet been disclosed). A few lines of an example of a misclassified non-vulnerable file (i.e., file considered as non-vulnerable in

the dataset, as no vulnerability in that file has been reported before; but it is classified as vulnerable by all classifiers) from the Linux Kernel source code is presented next (the whole file is not presented due to space constraints).

```
1 /*
2  * File Path: drivers/pcmcia/ds.c
3  * Software Metrics values:
4  * CountLineCode: 772
5  * SumCyclomatic: 208
6  * SumCyclomaticMod: 206
7  * SumCyclomaticStrict: 226
8  * SumEssential: 115
9  * CountPath: 2122
10 * FanIn: 399
11 * FanOut: 159.    */
12
13 static ssize_t field##_show (struct device *dev, struct
       device_attribute *attr, char *buf)    \
14 {                  \
15   struct pcmcia_device *p_dev = to_pcmcia_dev(dev);   \
16   return p_dev->test ? sprintf (buf, format, p_dev->field) : -
       ENODEV; \
17 }
```

As the values of the software metrics show (lines 3 to 11), the file is quite **complex**. To find out whether the file is indeed vulnerable or is a real false alarm, we performed a code review analysis using different static code analysis tools (SATs): CppCheck, FlawFinder and Rats (Brar and Kaur [2015]). We choose these tools since they are open source and are widely used to highlight different possible vulnerabilities within static C/C++ source code. For instance, *CppCheck* (https://cppcheck.sourceforge.io/) provides unique code analysis to detect bugs and focuses on detecting undefined behaviour and dangerous coding constructs, *FlawFinder* (https://dwheeler.com/flawfinder/) reports possible security weaknesses (i.e., flaws) sorted by risk level, and *RATS* (https://security.web.cern.ch/recommendations/en/codetools/rats.shtml) scans typical erros such as buffer overflows, and design flaws. Reported potential security problems are known as *severities*. The tools found several issues in the file, from which we could confirm two vulnerabilities. One of them can be seen in line 16 (i.e., *CWE-134 - Use of Externally-Controlled Format String*) where *sprintf* operation is used without checking the input value (**Missing Checking Input Value** bug according to ODC). Another one, which is not presented here, is a *CWE-120 - Buffer Copy without Checking Size of Input*, a classic buffer overflow issue.

3.3.5.3 Code Review

We run the same three SATs mentioned above (CppCheck, FlawFinder and Rats) on all files of each group (TPs, FNs, TNs, FPs). The results are summarized in Table 3.24. Each column shows the number of files of each group in which some sort of severity (not necessarily a security issue) is detected by each SAT tool. As shown, all SAT tools detected a higher percentage of files with severity in the TPs and FPs group (e.g., CppCheck: 84.4% and 87.5% respectively) than

in FNs and TNs (CppCheck: 39%% and 47.3% respectively). This again shows that files that are classified as vulnerable by all prediction models seem to be more untrustworthy than others.

Table 3.24: Static Code Analyzers results.

# Files		TP	FN	TN	FP
		1047	76	61803	2941
# Files with severities	Cppcheck	884 (84,4%)	39 (51,3%)	29205 (47,3%)	2572 (87,5%)
	Flawfinder	884 (84,4%)	42 (55,3%)	27467 (44,4%)	2520 (85,7%)
	RATS	564(53,9%)	14 (18,4%)	13041 (21,1%)	1465 (49,8%)

As the low effectiveness of SATs in detecting security vulnerabilities is quite well-known in the literature, we performed a manual code review to be more precise about the SATs results. Becauses this is a time-consuming task, we were not able to review all of the files. Instead, from each group (TP, FN, TN and FP), we randomly selected 15 files to be reviewed.

The main objectives of this code review are: *i)* to check whether the detected severities are the source of security issues and, if so, what kind of defect it is according to ODC bug classification; and *ii)* to verify whether the reported severities in the vulnerable files correspond to the exploited vulnerabilities that we have in the dataset. The results are presented in Table 3.25.

Table 3.25: Expert-based analysis results.

	TP	**FN**	**TN**	**FP**
# Files	15	15	15	15
# High sev.	35	49	33	85
# possible vuln.	2	9	8	10
# vulnerable files	2	5	3	4
ODC classification				
Checking - Missing	1	6	1	10
Algorithm - Wrong	-	3	1	-
Interface - Missing	1	-	-	-
Assignment - Wrong	-	-	6	-
Severity type				
fixed size local buffer	1	4	-	-
(format) sprintf	1	1	7	1
(crypto) crypt	-	3	-	-
(format) syslog	-	1	-	-
read	-	-	1	-
(buffer) strcpy	-	-	-	6
(buffer) sprintf	-	-	-	3

Observing the results presented in Table 3.25, we were able to confirmed our claim about the code units that are labeled as non-vulnerable. Out of the 30 non-vulnerable files, we could confirm 18 vulnerabilities in 7 files. Based on the ODC classification, we can see that most of the vulnerabilities detected by

SATs are caused by missing checking (e.g., input values), Wrong Assignment, and Wrong algorithm. The severity types identified by SATs in the case of confirmed vulnerabilities are limited to a list of operations that mostly manipulate buffers.

3.4 Summary

This chapter presented a set of experiments to study if software metrics and Machine Learning algorithms can be used for classifying vulnerable and non-vulnerable units of code. To do so, we divided the work in two steps: the prior included statistical correlation analysis, dimension reduction and feature selection, to understand the correlation between software metrics and security vulnerabilities (Section 3.2); the late focused on studying how discriminative software metrics and machine learning algorithms are for classifying vulnerable code units in different application scenarios (Section 3.3).

Considering the **statistical analysis** (that studied the interdependency between software metrics and the correlation of software metrics with the existence and number of vulnerabilities), the **dimention reduction** (to select different groups of software metrics) and the **feature selection** (where an heuristic search technique at the function and file levels to find the best subset of metrics for building a classification model), we can conclude that:

- There is a strong correlation between several project-level metrics and the number of reported vulnerabilities;
- We cannot clearly state if dimension reduction is (or not) helpful to achieve better performance, as the combination of metrics that leads to the best performance strongly depends on the Machine Learning algorithm. For example, Xboost achieves the best result when all metrics are used, while Decision Tree shows a better performance when the redundant file-level metrics are eliminated;
- It is possible to use a group of metrics to distinguish vulnerable and non-vulnerable units of code with a high level of accuracy. However, the best subset of predictive metrics may vary from one software system to another;
- For understanding the quality of software in terms of security, the function, file and project level metrics are complementary to each other.

There are, however, some limitations and threats to the validity that should be mentioned:

- Accuracy is used to evaluate the subset of metrics during the search process. Despite its popularity, accuracy does not show the false positive and negative rates of the classifier model, which are very important criteria for vulnerability detection tools. Also, a high level of accuracy is not very indicative performance metrics for imbalanced datasets. Thus, to deal with this search problem we should use several criteria, such as Recall, Informedness, F-Measure, and Markedness, in addition to accuracy,

to evaluate the subsets of metrics.

- To perform the heuristic search, we used the genetic algorithm combined with Random Forest. We do not claim that these techniques were the best choices, although they are highly recommended and commonly used in the literature to solve this kind of problems. We could use other techniques. Using other techniques and making a comparison between the results is left for future work.
- Configuring the parameters of the genetic algorithm is one of the main challenges that needs to be properly addressed to achieve high quality results. In this work, we performed an empirical calibration to choose the best values for these parameters. However, the calibration was done using the dataset of just one project (Apache https) and its results are used for all projects, which may not be adequate, mainly due to the fact that the calibration result is usually sensitive to dataset. The results achieved for Glibc are an evidence for the limitation associated with our calibration. Nevertheless, our calibration results allowed achieving the main objective of this work that is, the selection of the most predictive software metrics to distinguish vulnerable from non-vulnerable code units.

As for the study on whether **software metrics and machine learning can be used to distinguish vulnerable from non-vulnerable code units**, the main insights are (these respond to the Research Questions (RQs) in Section 3.3):

- Software metrics are not sufficient evidence of security issues to be used directly for building prediction models that can distinguish vulnerable code from non-vulnerable code in different application scenarios (**RQ1**);
- ML models created from software metrics are effective for security-critical applications (highly-critical and critical), in which the detection of vulnerabilities is of high priority. In contrast, a large number of false alarms make them useless for scenarios with low critical or non-critical systems (**RQ2**);
- There is no best ML algorithm for all possible contexts of the vulnerability prediction problem. In our study, RF and Xboost provided more precise models than other algorithms when data from the same project is used for training. In contrast, DT and Linear SVM build more generalizable models, thus providing a better estimation when data from a different project is used to evaluate the model (**RQ3**). Thus, it becomes difficult to generalize the results and apply to different types of software systems;
- To build prediction models with good performance, a relatively balanced (i.e., having an enough number of vulnerable code units to prevent overfitting), precisely labeled (i.e., assuring that the code labeled as non-vulnerable is free from any vulnerability), and highly representative (i.e., covering a vast range of software projects implemented in different languages) dataset is required. Building such a dataset is a difficult if not almost impossible endeavour.

We are aware that this experimental work also has limitations that need to be taken into account, and most of these threats to validity are related to the dataset used:

- All the projects in the dataset are implemented in C/C++, and each programming language has its own characteristics in terms of security. Consequently, some of the outcomes obtained from our analysis may not be representative for software implemented in other languages (e.g., Java).
- The source of information regarding the vulnerabilities in the projects is limited to security reports. Consequently, the functions and files without reported vulnerabilities are not necessarily flawless. To build the classifier model, we followed a supervised approach, which considers that our dataset is completely labeled. However, although the records with vulnerabilities are (reliable) labeled, but the rest can be seen as being unlabeled. This way, semi-supervised approaches should be studied as alternative choices for such cases, where it is not trivial to verify the label of all records due to the size of the dataset and complexity of the code.
- Although we used the well-known, commonly used, recommended, and representative machine learning algorithms, the number and diversity are still limited for a comprehensive analysis. Furthermore, the analysis for demonstrating the representativeness of random samples as well as the analysis performed for the understanding the impact of class distribution are done over the source code of a single project. This may have some implications on the results obtained with the other projects.

According to our observations and insights, we can conclude that **software metrics are not sufficient evidence of security issues to be used solely for building detection/prediction models that are able to distinguish vulnerable code from non-vulnerable code with good performance and low vulnerability removal cost**. Based on this conclusion, in the next chapters we will be focused on using software metrics not for predicting or detecting vulnerabilities but for assessing the trustworthiness of the code and warn the developers about their untrustworthy (insecure) code units.

Chapter 4

Trustworthiness Benchmarking using Software Metrics

Trustworthiness is a paramount concern for users and customers in the selection of a software solution, specially in the context of complex and dynamic environments. However, assessing and benchmarking trustworthiness (worthiness of software for being trusted) is a challenging task, mainly due to the variety of application scenarios (e.g., business-critical, safety-critical), the large number of determinative quality attributes (e.g., security, performance), and last, but foremost, due to the subjective notion of trust and trustworthiness.

This chapter takes trustworthiness as a measurable notion in relative terms based on security attributes and proposes a framework for the assessment and benchmarking of software units. The main goal is to build a trustworthiness assessment model based on security evidences (e.g., software metrics, code smells) that can be used as indicators of software quality. To do so, a trustworthiness score is assigned to each unit of code, calculated using the normalized value of relevant features (e.g., security evidences) and their impact on the precision of the classifier, which is measured by different variable importance measures (e.g., Mean Decrease Accuracy and Mean Decrease Gini). This relative trustworthiness score can then be used to compare and rank software elements.

To demonstrate the proposed framework, we assessed and ranked a number of files and functions of the Mozilla Firefox project based on their trustworthiness score (using software metrics as security evidences), and conducted a survey among several software security experts in order to validate the obtained rank. Results show that our framework is able to provide a sound ranking of the benchmarked software units.

The outline of this chapter is as follows. First, we present the trustworthiness benchmarking framework and discuss key aspects such as statistical analysis and normalization, relative importance of features, and trustworthiness assessment

process. In Section 4.2, we present an instantiation of the framework as a concrete example of trustworthiness assessment using software metrics as security evidences. Experimental results are discussed in Section 4.3, while validation and generalization aspects are discussed in Section 4.4. The chapter closes with a summary of the results.

4.1 Trustworthiness Benchmarking Framework

Figure 4.1 depicts the main components of the proposed framework, which results in the calculation of a trustworthiness score that allows comparing the trustworthiness of different software elements (e.g., functions, files). The trustworthiness score, used as the sole benchmarking criterion, is calculated based on *Security Evidences* (i.e., values of features) and their relative importance (i.e, weight of features), which are the main inputs for the benchmark. As we can observe in the figure, three phases are considered:

1. **Statistical Analysis and Data Normalization:** Since the features are measured using different scales, there is the need to normalize them to a common scale. To this end, firstly a statistical analysis should be performed on the input data to better understand the distribution of the values of each feature, and then, the data is normalized using Feature Scaling.
2. **Identifying Relative Importance of Features:** This phase consists of computing the weight of the features by studying how determinative each one is in the classification of vulnerable and non-vulnerable units of code.
3. **Trustworthiness Assessment:** Calculation of the trustworthiness of the software units under assessment. The output is a **trustworthiness score** that allows comparing the trustworthiness of different units.

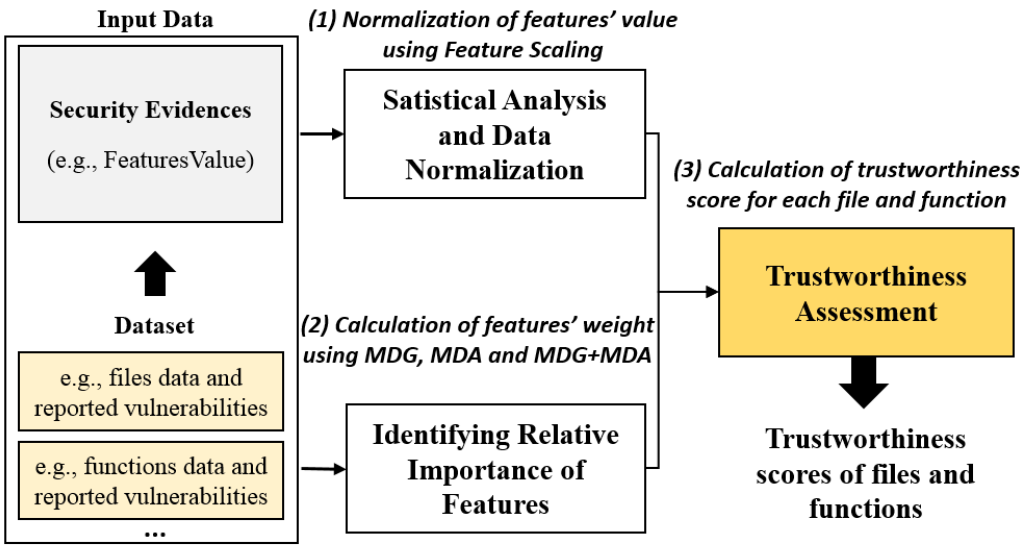


Figure 4.1: Instantiation of the Trustworthiness Benchmarking Framework.

The proposed framework is generic and easily applicable to different trustworthiness attributes. In fact, we can replace security attributes with any other trustworthiness attributes (e.g., performance) or add other attributes to the model. For this, we just need to identify the criteria for measuring the corresponding trustworthiness attributes (e.g., throughput, response time) and their relative importance.

4.1.1 Statistical Analysis and Normalization

To normalize the value of different features to a common scale, one should perform a statistical analysis on the input data to better understand the distribution of the values of each feature. The proposal is to calculate minimum, maximum, mean and quartiles values for each one. With this statistical information, we can simply observe if there are any cases with very large values that affect the normalization process. To eliminate this effect, outliers can be identified using a statistical method based on the Interquartile Range (IQR) (Walfish [2006]).

The IQR is the range between the first and the third quartiles (I$Q_3 - Q_1$). Using this method, any data value falling outside of the acceptable range, between the lower fence and upper fence, is considered to be an outlier:

$$\begin{aligned} AcceptableRange &= [LowerFence, UpperFence] \\ UpperFence(UF) &= Q_3 + 1.5 * IQR \\ LowerFence(LF) &= Max(Q_1 - 1.5 * IQR, 0) \\ IQR &= Q3 - Q1 \end{aligned} \tag{4.1}$$

Feature Scaling (Borkin et al. [2019]], a popular method to normalize the values of independent variables, should be used to bring the values of the features into a common range (between 0 and 1, in our work). As we are interested in characterizing trustworthiness, a higher score (1) should represent a more trustworthy code unit, thus we need to verify whether each individual feature has a direct or inverse relationship with the level of trustworthiness.

For this end and considering that we are going to use software metrics as security evidences in the framework instantiation, we can resort to sections 3.2.1 and 3.2.2 (statistical correlation analysis) to check the *partial dependency* between each selected metric and the existence of vulnerabilities.

In most cases, there is an obvious direct relationship, which means an inverse relationship with trustworthiness: the greater the value of the metric, the more likely to have a vulnerability, thus the less trustworthy. In a few cases, we have not seen an obvious direct or inverse relationship. This may seem a bit odd since these metrics are selected by the classifier as predictive metrics, but it happens because partial dependency of each metric is calculated individually and, as ex-

plained previously, a software metric that is irrelevant when considered individually might be highly significant when combined with other metrics. However, since we are going to normalize the values of the software metrics individually, we ended up concluding that the selected software metrics, have either an inverse relationship or an indifferent relationship.

Based on this relationship and given the *Acceptable Range* (see Equation 4.1), the value X for each software metric is scaled into the range [0,1] by using Equation 4.2.

$$X' = \begin{cases} 1 - \dfrac{X - LF}{UF - LF} & : X \text{ is in acceptable range} \\ 0 & : X \text{ is an outlier} \end{cases} \tag{4.2}$$

4.1.2 Relative Importance of Features

To identify the weight of the features (software metrics in our approach instantiation) for building the trustworthiness benchmarking framework, we can resort to Section 3.2.3 (feature selection analysis) and use the outputs of the random forest classifier to calculate two common variable importance measures, namely Mean Decrease Accuracy (MDA) and Mean Decrease Gini (MDG) (Calle and Urrea [2010]) for each software metric (although any other approaches to compute the relative importance of features can be used). In addition to these two measures, we also consider the MDAMDG joint measure, which previous research (Han et al. [2016]) has shown to be a robust one to identify the relative importance of the features used to build a classifier. These measures are explained next:

- **Mean Decrease Accuracy (MDA)** - shows how much the classifier's accuracy decreases by dropping a software metric from the list of features. MDA is normalized by the number of trees in the random forest. The greater the accuracy drops, the more significant the software metric.
- **Mean Decrease Gini (MDG)** - is a measure of how each software metric contributes to the homogeneity of nodes in the trees of the resulting random forest. Each time a particular software metric is used to split a node, the Gini coefficient for the child nodes are calculated and compared to that of the original node. Metrics that result in nodes with higher purity have a higher decrease in Gini coefficient.
- **MDAMDG** - is a joint measure whose value is the sum of MDA and MDG. This measure is used to calculate the total *importance score* of each software metric. The greater the total score, the more significant the software metric is.

Given the above explanation, three importance scores (i.e., MDA, MDG, and MDAMDG) are assigned to each evidence, resulting in three different weights. Each weight is calculated by normalizing the value of the corresponding importance score to a [0-1] range by considering the number of software metrics, so

that, for each score, the sum of the weights of all software metrics is 1.

4.1.3 Trustworthiness Assessment

To obtain the trustworthiness score for each unit of code under evaluation, we propose using the **Simple Additive Weighting (SAW)**, which is a commonly used Multi-Criteria Decision-Making method (Aruldoss et al. [2013]). According to the SAW method, the score of each given alternative (i.e., a code unit in this context) is calculated as the weighted sum of the quality of that alternative (i.e., the code quality in this context) on each attribute (i.e., features) (Velasquez and Hester [2013]). For example, let us assume several functions for being reviewed from a security perspective. Reviewers should evaluate the trustworthiness of each function and assign it a score (in the range of 0 to 1). The evaluation should be performed based on two metrics, e.g., function visibility and number of input parameters, with different importance (e.g., function visibility composes 80% of the final score and the number of input parameters composes 20% of the final score). As a result, each function receives a final trustworthiness score from each reviewer. For one function, if a given reviewer gives 0.9 out of 1 for the first metric and 0.5 out of 1 for the second metric, the final trustworthiness score of the function, calculated as the weighted sum of the two metrics, would be 0.9 * 0.8 + 0.5 * 0.2 = 0.82.

Based on this concept, to obtain the trustworthiness score for each unit of code under evaluation, we need to calculate the sum of the product between the normalized value of each feature (M) and its associated weight (W), as shown in Equation 4.3.

$$Trustworthiness\ Score = \sum_{i=1}^{n} M_i W_i \tag{4.3}$$

The index n represents the total number of features used. Note that, this score is a **relative** measure of trustworthiness that should only be used for comparison purposes and not as an absolute measure of security or trustworthiness. The assumption is that, a trustworthiness level refers to the extent to which a piece of software can be trusted and the trustworthiness of a code unit can be determined by the combination of pieces of evidence of software security, showing that it is trustworthy (Neto and Vieira [2011a]).

Since we calculate three different weights for each feature (MDA, MDG, and MADMDG), we get three different trustworthiness scores for each code unit. These scores can finally be used to compare/rank a set code units under evaluation.

4.2 Framework Instantiation

In this section, we present a concrete instantiation of the framework to build trustworthiness assessment models, taking software metrics as evidences of security. For this, we work on top of files and functions of the Mozilla Firefox data,

since it is a quite large and well-known software project, with a large number of known vulnerabilities, a fundamental aspect to achieve more accurate results. Table 4.1 presents some information about the project. As mentioned before, the dataset has been updated over time (either because there were changes in the dataset or refinements in the data search). Because of this, we can observe a slight difference in the number files and functions of Mozilla Firefox project comparing with the dataset used in the Section 3.1. It is important to recall that the approach is generic and applicable to any other software project, as far as the source code is available for extracting the files and functions data.

Table 4.1: Summary of the Mozilla Firefox Project.

	Total	Vulnerable	% Vulnerable	# Software Metrics
# Files	28927	830	2,87%	21
# Functions	614422	2107	0,34%	15

The software metrics used as security attributes, for both files and functions, were identified using the feature selection presented in Section 3.2.3. Those metrics were selected as the best subset of metrics for building a classifier model (using Mozilla Firefox project data) allowing to distinguish vulnerable pieces of code from non-vulnerable ones. A total of 21 file-level metrics and 15 function-level metrics were used, as shown in tables 3.11 and 3.16, respectively. These input data represent the **trustworthiness evidences** in our benchmarking process.

To demonstrate our benchmarking framework, we selected several files and functions from the source code of the software project under study (i.e., Mozila Firefox) and ranked them using the trustworthiness scores obtained. The process is depicted in Figure 4.2 and includes two main steps: *i) qualify and select the files and functions for validation*, and *ii) rank the selected files and functions based on the trustworthiness score calculated.*

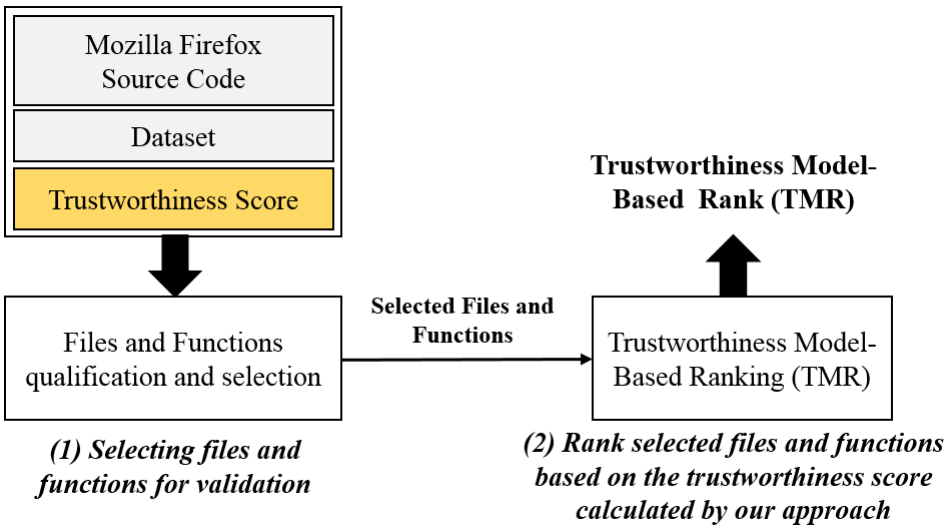


Figure 4.2: Benchmarking Process.

To **select a meaningful list of files and functions** from the Mozilla Firefox codebase, we divided the files/functions in two groups: with and without known vulnerabilities. This was done, first because the files and functions with vulnerabilities do not qualify for trustworthiness evaluation (they cannot be trusted). Second, we want to eliminate any possible effect of the existence of known vulnerabilities (in files and functions) on the judgments of the experts, which will be used later to validate the output of our approach. However, for the sake of completeness, we repeat the whole process, including experiments and analyses, for this group of files/functions (with vulnerabilities) to validate the proposed approach for this specific case (i.e., to understand its usefulness to compare code with known vulnerabilities).

We selected 5 files and 5 functions from each group/category. We selected only 5 to make the validation process feasible, since experts were asked to compare each pair of files/function separately (i.e., a total of 90 comparisons are needed for only 5 files and 5 functions). The procedure was as follows:

- **First selection based on the value of individual software metrics** - To guarantee the diversity of our selection, we first sorted the files/-functions according to the individual software metrics value (21 software metrics for file level and 15 software metrics for function level). For each metric, we then selected the files/functions having the minimum, lower fence, mean, upper fence and maximum values, resulting in the selection of 105 files (5 files selected for each of the 21 file-level metrics) and 75 functions (5 functions selected for each of 15 file-level metrics). A random choice of one was made in the cases where several files/functions had the same value.
- **Cleaning** - In this step, we first removed duplicated files/functions and then removed the files/functions in which all software metrics have a normalized value equal to 0, as in this case we cannot calculate a trustworthiness scores.
- **Final selection of files and functions** - We sorted the remaining 71 files and 54 functions according to the **trustworthiness score** calculated using the MDAMDG weight, a joint measure of both MDA and MDG (in most cases the order of importance of the metrics given by MDA and MDG is the same as of MDAMDG). Then we selected 5 non-vulnerable files ($File_A$, $File_B$, $File_C$, $File_D$, $File_E$), 5 vulnerable files ($File_F$, $File_G$, $File_H$, $File_I$, $File_J$), 5 non-vulnerable functions ($Func_A$, $Func_B$, $Func_C$, $Func_D$, $Func_E$) and 5 vulnerable functions ($Func_F$, $Func_G$, $Func_H$, $Func_I$, $Func_J$) with different levels of trustworthiness score ranging from the minimum to maximum. It is worth noting that the selection process is only based on the metrics and not based on the detailed analysis of the source code.

As final step, the selected files/functions where ranked based on the three trustworthiness scores (MDG, MDA, MDAMDG) and the results were analysed. Furthermore, in order to validate the ranking provided by the proposed trustworthiness benchmarking approach we conducted an expert-based ranking. This

ranking was compared with the one obtained by our approach, both for files and functions without vulnerabilities and files and functions with vulnerabilities, as will be discussed next.

4.3 Assessment and Results

In this section, we present and analyze the results obtained, including the **statistical analysis and normalization**, the **relative importance of software metrics** and the **trustworthiness assessment results**.

4.3.1 Statistical Analysis and Normalization

As mentioned before, since the software metrics are measured using different scales, there is a need to normalize them to a common scale. To this end, a statistical analysis needs to be performed on the input data to better understand the distribution of the values of each software metric.

The statistical data obtained for the file and function level metrics are summarized in Table 4.2 and Table 4.3, respectively. Here, we intend to analyze the distribution of the values of the software metrics at both file and function levels, in order to identify the outliers that affect the normalization process. As previously explained, the outliers are identified by calculating the acceptable range of values based on IQR ($IQ_3 - Q_1$).

Table 4.2: Statistical data of file-level metrics.

No.	Software metrics	Lower Fence	Min.	Q1	Mean	Q3	Upper Fence	Max.	# outliers
1	**FanOut**	0	0	13	102,49	111	258	10136	2541
2	**CountPath**	0	0	8	29173420,8	213	520,5	8048563425	5118
3	**SumMaxNesting**	0	0	2	19,92	21	49,5	1893	2536
4	**FanIn**	0	0	8	108,76	85	200,5	129882	2743
5	**SumCyclomaticStrict**	0	0	9	88,00	87	204	12650	3121
6	**SumCyclomaticModified**	0	0	8	74,67	74	173	9129	3131
7	**CountDeclFunction**	0	0	3	18,05	20	45,5	1281	2616
8	**AvgFanOut**	0	0	3	5,72	7	13	190	1345
9	**SumCyclomatic**	0	0	8	79,16	79	185,5	9931	3109
10	**CountLineCodeDecl**	0	0	14	100,76	98	224	50216	2957
11	**CountSemicolon**	0	0	22	185,47	187	434,5	28029	3014
12	**AvgLineCode**	0	0	7	18,64	22	44,5	1177	1849
13	**AvgMaxNesting**	0	0	0	1,08	1	2,5	10	1860
14	**AvgCyclomaticStrict**	0	0	2	5,10	6	12	649	1753
15	**AvgCyclomaticModified**	0	0	2	4,25	5	9,5	428	1919
16	**RatioCommentToCode**	0	0	0,12	0,60	0,5	1,07	56	3019
17	**AltAvgLineBlank**	0	0	0	2,61	3	7,5	275	1807
18	**CBO**	0	0	0	8,49	0	0	7958	5600
19	**AvgEssential**	0	0	1	2,15	2	3,5	236	3713
20	**CountStmtEmpty**	0	0	0	5,66	2	5	4794	4274
21	**LCOM**	0	0	0	95,43	0	0	108941	3831

Table 4.3: Statistical data of function-level metrics.

No.	Software metrics	Lower Fence	Min.	Q1	Mean	Q3	Upper Fence	Max.	# outliers
1	**CountLine**	0	0	5	22,7	24	52,5	5552	58908
2	**CountOutput**	0	0	2	5,9	7	14,5	665	50150
3	**CyclomaticModified**	0	0	1	4,2	4	8,5	1798	66187
4	**CountLineCode**	0	0	5	17,8	19	40	3028	56943
5	**CountSemicolon**	0	0	1	10,1	10	23,5	3316	59143
6	**CountStmtExe**	0	0	1	10,5	11	26	3991	53070
7	**CountLineCodeExe**	0	0	1	11,1	12	28,5	2347	54652
8	**CountStmtDecl**	0	0	0	2,2	2	5	663	65322
9	**CountLineComment**	0	0	0	2,0	1	2,5	2181	103550
10	**Knots**	0	0	0	6,1	2	5	83057	71240
11	**MinEssentialKnots**	0	0	0	4,7	2	5	82998	57705
12	**MaxEssentialKnots**	0	0	0	4,7	2	5	83011	58694
13	**Essential**	0	0	1	2,5	3	6	1313	46769
14	**CountStmtEmpty**	0	0	0	0,2	0	0	2178	56418
15	**CountLineInactive**	0	0	0	0,6	0	0	1465	31989

We can observe that all metrics have a minimum value of 0 for both file and function levels, and considering the amplitude of values, we can state that the greatest range of values is observed from the Q3 onwards. The same is to say that a smaller variation of values is observed in the first 75% of the ordered elements (files or functions) of the sample. Based on this simple analysis, we observed that there are cases with very large values that affect the normalization process (e.g., for the *FanIn* software metric, the maximum value existing in the dataset is 129882 while the mean value is 108). Also, the results of our statistical analysis show that, for each software metric, different number of values are considered to be outliers (refer to the last column of tables 4.2 and 4.3).

As an example, Table 4.4 presents the real (a) and normalized (b) values for 4 out of 21 file-level metrics for the five files selected/qualified for validation without known vulnerabilities (labeled as nv), as we can see in Label column. When the normalized value is 0 that means that the original value is higher than the upper fence and is considered an outlier (e.g., the values of CountLineCodeDecl, SumCyclomatic, CountPath and FanIn for file 5 or the values of CountLineCodeDecl, SumCyclomatic and CountPath for file 4). On the other hand, when the normalized value is 1, it means that the original value is always 0 (e.g., the value of Sum Cyclomatic, Count Path and FanIn for file 1). For the remaining cases, the result of Equation 4.2 is used.

Examples of the normalization results for function-level metrics are presented in Table 4.5. The table presents the real (a) and normalized (b) values for 4 out of 15 function-level metrics for the five functions selected/qualified for validation without known vulnerabilities.

Table 4.4: Example of the real (a) and normalized (b) values of four software metrics for five Mozilla Firefox's files.

(a) File-level dataset of Mozilla Firefox project

No.	Label	CountLineCodeDecl	SumCyclomatic	CountPath	FanIn
1	nv	147	0	0	0
2	nv	16	23	117	5
3	nv	16	248	24	68
4	nv	772	500	6338	71
5	nv	292	4793	4542149064	241

(b) Normalized File-level dataset of Mozilla Firefox project

No.	Label	CountLineCodeDecl	SumCyclomatic	CountPath	FanIn
1	nv	0,344	1,000	1,000	1,000
2	nv	0,929	0,876	0,775	0,975
3	nv	0,929	0,000	0,954	0,661
4	nv	0,000	0,000	0,000	0,646
5	nv	0,000	0,000	0,000	0,000

Table 4.5: Example of the real (a) and normalized (b) values of four software metrics for five Mozilla Firefox's functions.

(a) Function-level dataset of Mozilla Firefox project

No.	Label	CountLineCodeDecl	CountOutput	CyclomaticModified	Essential
1	nv	0	2	0	0
2	nv	18	10	2	1
3	nv	6	8	8	3
4	nv	33	9	6	6
5	nv	35	14	15	7

(b) Normalized Function-level dataset of Mozilla Firefox project

No.	Label	CountLineCodeDecl	CountOutput	CyclomaticModified	Essential
1	nv	1,000	0,862	1,000	1,000
2	nv	0,550	0,310	0,765	0,833
3	nv	0,850	0,448	0,059	0,500
4	nv	0,175	0,379	0,294	0,000
5	nv	0,125	0,034	0,000	0,000

4.3.2 Relative Importance of Software Metrics

As mentioned before, the software metrics considered in this study are the ones selected by the feature selection technique discussed in Section 3.2.3 as the best subset of metrics for building the most accurate classifier over the Mozilla Firefox dataset. With these, we calculated the three weights for each metric using Mean Decrease Gini (MDG), Mean Decrease Accuracy (MDA), and MDAMDG as discussed in Section 4.1.2.

The results obtained for the file and function levels are presented in Table 4.6

and Table 4.7, respectively. The metrics are sorted based on the MDAMDG value. As shown in both tables, the weights calculated using MDG, MDA, and MDAMDG are different, but in most cases the order of importance of the metrics given by MDA and MDG is the same as of MDAMDG. The exceptions are marked in red.

The relative importance of software metrics used to calculate the trustworthiness score are presented in the last 3 columns of tables 4.6 and 4.7, for files and functions, respectively. These weights are the ones used as input to compute the trustworthiness scores of files and functions of Mozilla Firefox project. Note that, the sum of the weights of all software metrics is always 1 (e.g., sum of the weights for MDAMDG_W column).

Table 4.6: File-level metric weights.

File-level Metrics	MDG	MDA	MDAMDG	MDG_W	MDA_W	MDAMDG_W
FanOut	100,27	38,58	138,85	0,102	0,081	0,095
CountPath	95,55	41,45	137,00	0,098	0,087	0,094
SumMaxNesting	101,34	31,76	133,10	0,104	0,067	0,091
FanIn	88,84	32,74	121,58	0,091	0,069	0,084
SumCyclomaticStrict	79,38	26,98	106,37	0,081	0,057	0,073
SumCyclomaticModified	69,57	27,83	97,40	0,071	0,058	0,067
CountDeclFunction	68,96	28,04	97,01	0,070	0,059	0,067
AvgFanOut	61,08	31,76	92,84	0,062	0,067	0,064
SumCyclomatic	61,70	26,11	87,81	0,063	0,055	0,060
CountLineCodeDecl	52,08	22,39	74,47	0,053	0,047	0,051
CountSemicolon	47,89	20,87	68,77	0,049	0,044	0,047
AvgLineCode	25,23	19,68	44,91	0,026	0,041	0,031
AvgMaxNesting	22,95	19,83	42,78	0,023	0,042	0,029
AvgCyclomaticStrict	21,37	18,80	40,17	0,022	0,039	0,028
AvgCyclomaticModified	19,07	18,38	37,45	0,019	0,039	0,026
RatioCommentToCode	17,50	15,20	32,71	0,018	0,032	0,022
AltAvgLineBlank	10,85	13,02	23,87	0,011	0,027	0,016
CBO	9,62	14,06	23,68	0,010	0,030	0,016
AvgEssential	11,00	9,50	20,51	0,011	0,020	0,014
CountStmtEmpty	9,05	9,63	18,68	0,009	0,020	0,013
LCOM	5,42	9,54	14,96	0,006	0,020	0,010

As we can observe, the file-level metrics with greater relative importance are *FanOut* and *CountPath* with 0.095 and 0.094 MDAMDG weight value, respectively. On the other hand, the file-level metrics with lowest relative importance value are *CountStmtEmpty* and *LCOM* with 0.013 and 0.010 MDAMDG weight value (see column MDAMDG_W of Table 4.6). On the other hand, *CountLine* and *CountOutput* are the metrics with higher weights (0.134 and 0.117 for MDAMDG_W) and *CountStmtEmpty* and *CountLineInactive* are the metrics with lower weights (0.017 and 0.012 for MDAMDG_W) for funtion-level metrics, as we can verify in Table 4.7.

Table 4.7: Function-level metric weights.

Function-level Metrics	MDG	MDA	MDAMDG	MDG_W	MDA_W	MDAMDG_W
CountLine	205,62	37,72	243,34	0,145	0,094	0,134
CountOutput	165,97	47,77	213,74	0,117	0,119	0,117
CyclomaticModified	151,40	41,79	193,19	0,107	0,104	0,106
CountLineCode	146,21	26,65	172,86	0,103	0,066	0,095
CountSemicolon	130,89	29,84	160,73	0,092	0,074	0,088
CountStmtExe	127,00	31,83	158,83	0,089	0,079	0,087
CountLineCodeExe	126,13	24,66	150,79	0,089	0,061	0,083
CountStmtDecl	98,62	32,67	131,29	0,069	0,081	0,072
CountLineComment	86,10	43,43	129,53	0,061	0,108	0,071
Knots	54,31	22,46	76,77	0,038	0,056	0,042
MinEssentialKnots	30,25	17,69	47,94	0,021	0,044	0,026
MaxEssentialKnots	28,77	17,24	46,02	0,020	0,043	0,025
Essential	29,12	15,57	44,70	0,021	0,039	0,025
CountStmtEmpty	21,01	9,34	30,35	0,015	0,023	0,017
CountLineInactive	18,25	4,04	22,29	0,013	0,010	0,012

4.3.3 Results and Discussion

The trustworthiness scores (T Score) of the non-vulnerable files ($File_A$, $File_B$, $File_C$, $File_D$, $File_E$) and functions ($Func_A$, $Func_B$, $Func_C$, $Func_D$, $Func_E$) calculated by our trustworthiness model for three different weights are presented in tables 4.8 and 4.9, respectively. In both tables, the files and functions are sorted from the most trustworthy to the least trustworthy. It is interesting to notice that, independently of the weights (MDG_W, MDA_W and MDAMDG_W) used to calculate the trustworthiness score, the order of the files is always the same.

Table 4.8: Ranking of files without known vulnerabilities.

MDG_W		MDA_W		MDAMDG_W	
Files	T Score	Files	T Score	Files	T Score
FileC	0,930	FileC	0,886	FileC	0,916
FileA	0,757	FileA	0,653	FileA	0,723
FileE	0,500	FileE	0,444	FileE	0,482
FileD	0,262	FileD	0,211	FileD	0,245
FileB	0,051	FileB	0,052	FileB	0,051

The ranking for files is: $File_C > File_A > File_E > File_D > File_B$, where the symbol $>$ means *more trustworthy than*. We can observe that $File_C$ is considered the most trustworthy file with a trustworthiness score of 0.916 (MDGMDA_W). On the other hand, $File_B$ is the least trustworthy one with a trustworthiness score of 0.051 (MDGMDA_W).

Table 4.9: Ranking of functions without known vulnerabilities

MDG_W		MDA_W		MDAMDG_W	
Functions	**T Score**	**Functions**	**T Score**	**Functions**	**T Score**
FuncE	0,956	**Func**E	0,950	**Func**E	0,955
FuncD	0,746	**Func**D	0,776	**Func**D	0,753
FuncB	0,501	**Func**B	0,467	**Func**B	0,493
FuncA	0,232	**Func**A	0,203	**Func**A	0,226
FuncC	0,043	**Func**C	0,029	**Func**C	0,040

The ranking for the functions is also the same, independently of the weight used: $Func_E > Func_D > Func_B > Func_A > Func_C$. Therefore, $Func_E$ is considered the most trustworthy function with a trustworthiness score of 0.955 (MDAMDG_W), while $Func_C$ is the least trustworthy function with a trustworthiness score of 0.040 (MDAMDG_W).

Regarding the results for vulnerable units of code (refers to tables 4.10 and 4.11 for files and functions, respectively), the order of files, independently of the weights (MDG_W, MDA_W and MDGMDA_W), is $File_I > File_G > File_J > File_F > File_H$. Thus, $File_I$ is considered the most trustworthy file with a trustworthiness score of 0.937 (MDGMDA_W). On the other hand, $File_H$ is the least trustworthy one with a trustworthiness score of 0.046 (MDGMDA_W). For functions, the order is $Func_G > Func_J > Func_F > Func_H > Func_I$, also independently of the weights. This means that, $Func_G$ is considered the most trustworthy function with a trustworthiness score of 0.950 (MDGMDA_W) while $Func_I$ is the least trustworthy function with a trustworthiness score of 0.028 (MDGMDA_W).

Table 4.10: Ranking of files with vulnerabilities.

MDG_W		MDA_W		MDAMDG_W	
Files	**T Score**	**Files**	**T Score**	**Files**	**T Score**
FileI	0,951	**File**I	0,908	**File**I	0,937
FileG	0,790	**File**G	0,732	**File**G	0,771
FileJ	0,506	**File**J	0,487	**File**J	0,500
FileF	0,240	**File**F	0,277	**File**F	0,252
FileH	0,045	**File**H	0,048	**File**H	0,046

Table 4.11: Ranking of functions with vulnerabilities

MDG_W		MDA_W		MDAMDG_W	
Functions	**T Score**	**Functions**	**T Score**	**Functions**	**T Score**
FuncG	0,951	**Func**G	0,945	**Func**G	0,950
FuncJ	0,770	**Func**J	0,763	**Func**J	0,768
FuncF	0,492	**Func**F	0,520	**Func**F	0,498
FuncH	0,258	**Func**H	0,318	**Func**H	0,271
FuncI	0,028	**Func**I	0,027	**Func**I	0,028

4.4 Validation and Generalization

In order to validate the ranking provided by the proposed trustworthiness benchmarking approach, we conducted an **expert-based ranking**. This ranking was compared with the **trustworthiness model-based ranking** (the ranking obtained by our approach), for both files/functions without vulnerabilities and files/functions with vulnerabilities.

After selecting the files and functions for validation, we asked twelve experts to rank the files and functions based on their *perceived trustworthiness* by providing them with the source code of the files/functions and the corresponding software metrics. In practice, the experts, with experience and interest in the security area (4 PhD students and 8 professors), working in different universities in different countries (Portugal, Italy, Brazil, Belgium), were asked to do a comparison between all the different pairs of files/functions. It is important to mention that all experts are researchers in the area of secure software engineering, some of which having more than 10 years of experience.

The experts were provided with the source code of the files/functions and the available information regarding the individual software metrics. They were asked to choose the more trustworthy file/function for each possible pair (a total of 90 pairwise comparisons) and assign a value indicating how much trustworthy a file/function is in comparison to another. For this, we defined four different levels: *much more trustworthy* (3), *more trustworthy* (2), *a little more trustworthy* (1) or *non-differentiable* (0). We believe that defining more levels would highly complicate the judgment process for the experts, thus not bringing any added value. For supporting further calculations, we transformed these values into quantitative ones, as shown in Table 4.12, according to the *Fundamental Scale of Absolute Numbers for Pairwise Comparison* (Martinez et al. [2014]).

Table 4.12: Absolute numbers in pairwise comparison.

Definition	Description	Intensity
Equal	Non-differentiable	1
Moderate	Little more trustworthy	3
Strong	More trustworthy	5
Very strong	Much more trustworthy	7

Figure 4.3 depicts the whole process of validation. As described, we start by the **analysis of the experts' responses** where we evaluate the consistency of the responses and perform a transitivity reduction. Then we calculate the **individual and aggregated rankings** of selected files and functions of the Mozilla Firefox project using the Row Geometric Mean Method (prioritization method). Finally, we compare the results of our trustworthiness model-based ranking and the ranking of the experts.

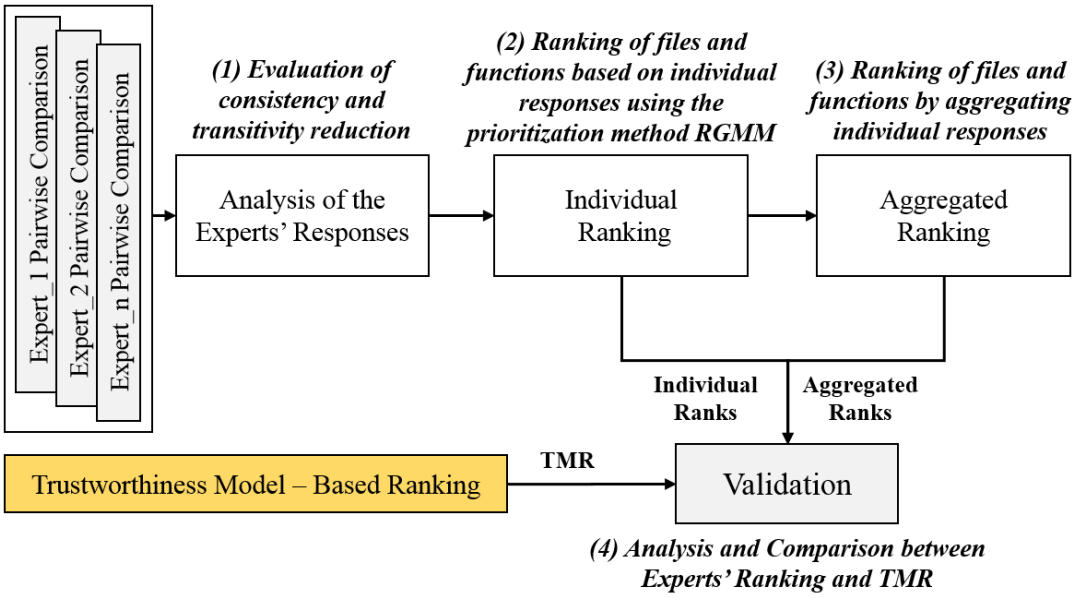


Figure 4.3: Validation process.

4.4.1 Responses of the Experts

We started by analyzing the responses of the experts to find inconsistencies. An inconsistency occurs when an expert chooses A rather than B, B rather than C, but C rather than A as the more trustworthy file/function, while based on the two first statements, A must be more trustworthy than C. For this, we generated a complete graph to summarize the responses, and performed transitivity reduction to help identifying such inconsistencies. Inconsistent responses were decided to be eliminated from the validation process in order to avoid the effect of contradictory responses on the aggregated ranking of files/functions.

The analysis of the responses resulted in the exclusion of 3 responses/experts (for both files and functions) from the whole process. Thus, 9 responses remained for further analysis.

The pairwise comparison of files and functions made by the remaining experts are presented in Table 4.13 and Table 4.14. Each row presents the comparisons carried out by one expert and the columns show the judgment made between each pair of files and functions. For example, in the first column of Table 4.13, C - A means that $File_C$ is being compared against $File_A$, and expert 1 (first row) chooses $File_A$ rather than $File_C$, but indicates 1, which means that $File_A$ is *a little more trustworthy* than $File_C$.

Observing tables 4.13 and 4.14, we can verify that:

- For expert 1, file A is a *little more trustworthy* than file C. However, for all other experts that compared these files, file C was classified as more trustworthy than files A, E, D and B (with different intensity level);
- In general, file C and file A are the most trustworthy, considering the opinion of the software security experts involved in the survey. On the

other hand, file B is the less trustworthy, since it is only selected three times as the more trustworthy, when compared with file D (expert 1, expert 5 and expert 6);

- Function E has been classified as more trustworthy than D, B, A and C functions (with different intensity level), except for the cases where the expert was unable to differentiate them (experts 4, 6, 7 and 9 were not able to differentiate between function E and D);
- In general, functions E and D are the most trustworthy considering the opinion of the experts. Functions A and C are the less trustworthy.

Table 4.13: Responses of the experts for files.

Files	C - A	C - E	C - D	C - B	A - E	A -D	A -B	E -D	E - B	D - B
Expert 1	A, 1	C, 1	C, 3	C, 1	A, 3	A, 2	A, 1	E, 2	E, 1	B, 2
Expert 2	C, 3	C, 3	C, 3	C, 3	A, 1	A, 3	A, 3	E, 3	E, 3	D, 2
Expert 3	C, 1	C, 1	C, 1	C, 3	A, 1	0	A, 3	0	E, 3	D, 3
Expert 4	0	C, 1	C, 2	C, 3	A, 1	A, 2	A, 3	0	E, 3	D, 1
Expert 5	C, 1	C, 2	C, 3	C, 3	A, 1	A, 2	A, 3	E, 1	E, 2	B, 1
Expert 6	0	C, 2	C, 3	C, 2	A, 2	A, 3	A, 2	E, 1	E, 1	B, 1
Expert 7	C, 3	C, 3	C, 3	C, 3	A, 2	A, 2	A, 2	E, 2	E, 2	0
Expert 8	C, 1	C, 3	C, 3	C, 3	A, 2	A, 2	A, 2	E, 1	E, 1	D, 1
Expert 9	C, 1	C, 3	C, 2	C, 3	A, 2	A, 1	A, 2	0	E, 1	D, 1

Table 4.14: Responses of the experts for functions.

Functions	E - D	E - B	E - A	E - C	D - B	D - A	D - C	B - A	B - C	A - C
Expert 1	E, 2	E, 1	E, 3	E, 3	D, 2	D, 1	D, 1	B, 2	B, 2	0
Expert 2	E, 3	E, 3	E, 3	E, 3	D, 2	D, 2	D, 3	B, 1	B, 1	A, 1
Expert 3	E, 3	E, 3	E, 3	E, 3	B, 1	D, 2	D, 2	B, 2	B, 2	0
Expert 4	0	E, 1	E, 1	E, 2	D, 1	D, 1	D, 2	0	B, 1	0
Expert 5	E, 1	E, 2	E, 3	E, 3	D, 1	D, 2	D, 3	B, 1	B, 2	A, 1
Expert 6	0	E, 3	0	E, 2	D, 3	0	D, 2	A, 3	C, 1	A, 1
Expert 7	0	E, 3	E, 3	E, 3	D, 1	D, 1	D, 1	A, 1	C, 1	A, 1
Expert 8	E, 1	E, 3	E, 3	E, 2	D, 3	D, 3	D, 1	A, 2	C, 2	C, 1
Expert 9	0	E, 3	E, 3	E, 3	D, 3	D, 3	D, 3	A, 2	C, 1	A, 1

4.4.2 Individual and Aggregated Ranking

Based on the pairwise comparisons, we calculated individual and aggregated ranks for files and functions. There are two possible methods for calculating the individual ranks: the Eigenvalue Method (EM) and the Row Geometric Mean Method (RGMM) (Crawford and Williams [1985]). However, it is shown in Dong et al. [2008] that the difference in the output of these two methods is meaningless, thus we choose **Row Geometric Mean Method**, which requires less computation power.

To calculate the integrated rank of the files and functions, there are also two known methods: the Aggregation of Individual Judgments (AIJ) (Dong et al. [2010a]) and the Aggregation of Individual Priorities (AIP). As shown in Dong et al. [2010a], when RGMM is used to calculate individual rank, the output of both methods is equivalent. Thus, in this work we use the **Aggregation of Individual Judgments**.

The individual and aggregated rankings obtained for non-vulnerable files and non-vulnerable functions for Mozilla Firefox project are presented in Table 4.15 and Table 4.16, respectively.

Table 4.15: Individual & aggregated rankings of non-vulnerable files.

Expert 1 Rank		**Expert 2 Rank**		**Expert 3 Rank**		**Expert 4 Rank**		**Expert 5 Rank**	
FileA	0,458	**FileC**	0,563	**FileC**	0,427	**FileC**	0,366	**FileC**	0,497
FileC	0,266	**FileA**	0,218	**FileA**	0,221	**FileA**	0,366	**FileA**	0,270
FileE	0,135	**FileE**	0,141	**FileD**	0,177	**FileE**	0,137	**FileE**	0,133
FileB	0,103	**FileD**	0,051	**FileE**	0,142	**FileD**	0,094	**FileB**	0,058
FileD	0,037	**FileB**	0,027	**FileB**	0,032	**FileB**	0,036	**FileD**	0,042
Expert 6 Rank		**Expert 7 Rank**		**Expert 8 Rank**		**Expert 9 Rank**		**Agg. Rank**	
FileC	0,387	**FileC**	0,581	**FileC**	0,521	**FileC**	0,513	**FileC**	0,408
FileA	0,387	**FileA**	0,218	**FileA**	0,274	**FileA**	0,261	**FileA**	0,310
FileE	0,112	**FileE**	0,114	**FileE**	0,099	**FileD**	0,099	**FileE**	0,171
FileB	0,072	**FileD**	0,044	**FileD**	0,064	**FileE**	0,084	**FileD**	0,065
FileD	0,041	**FileB**	0,044	**FileB**	0,041	**FileB**	0,043	**FileB**	0,046

Table 4.16: Individual & aggregated rankings of non-vulnerable functions.

Expert 1 Rank		**Expert 2 Rank**		**Expert 3 Rank**		**Expert 4 Rank**		**Expert 5 Rank**	
FuncE	0,521	**FuncE**	0,579	**FuncE**	0,586	**FuncE**	0,348	**FuncE**	0,497
FuncD	0,216	**FuncD**	0,232	**FuncB**	0,198	**FuncD**	0,348	**FuncD**	0,270
FuncB	0,154	**FuncB**	0,093	**FuncD**	0,128	**FuncB**	0,131	**FuncB**	0,133
FuncA	0,055	**FuncA**	0,060	**FuncA**	0,044	**FuncA**	0,105	**FuncA**	0,065
FuncC	0,055	**FuncC**	0,036	**FuncC**	0,044	**FuncC**	0,069	**FuncC**	0,035
Expert 6 Rank		**Expert 7 Rank**		**Expert 8 Rank**		**Expert 9 Rank**		**Agg.Rank**	
FuncE	0,305	**FuncE**	0,467	**FuncE**	0,488	**FuncE**	0,405	**FuncE**	0,458
FuncD	0,305	**FuncD**	0,281	**FuncD**	0,284	**FuncD**	0,405	**FuncD**	0,303
FuncA	0,275	**FuncA**	0,123	**FuncC**	0,130	**FuncA**	0,099	**FuncA**	0,094
FuncC	0,079	**FuncC**	0,079	**FuncA**	0,066	**FuncC**	0,058	**FuncB**	0,082
FuncB	0,037	**FuncB**	0,051	**FuncB**	0,031	**FuncB**	0,034	**FuncC**	0,063

As we can observe, four out of nine experts obtained the same complete ranking of files: $File_C > File_A > File_E > File_D > File_B$ (identified in green). This is the same rank obtained by the aggregation of individual judgments for files. In the obtained rank of functions, four out of nine experts have the same complete individual ranking: $Func_E > Func_D > Func_B > Func_A > Func_C$ (identified in green). In turn, the aggregated rank of functions (presented in Table 4.16)

is slightly different: $Func_E > Func_D > Func_A > Func_B > Func_C$. However, we can verify that $Func_E$ and $Func_D$ are considered the most trustworthy for eight out of nine experts.

The aggregated ranking in Table 4.15 (files) is exactly equal to the one computed by our benchmarking approach: $File_C > File_A > File_E > File_D > File_B$. Considering the individual rankings from the experts, 4 out of 9 experts ranked the files in that exactly same order. In the other cases, the experts partially indicate the same order (e.g., $File_C > File_A > File_E > File_B$ appears in 4 responses) and indicate a reverse order for the files that are close to each other in terms of their trustworthiness score ($File_D$ and $File_B$, with 0.245 and 0.051 MDAMDG scores, respectively). These results, on one hand, show that the inconsistency analysis and exclusion of the inconsistent responses were effectively done. On the other hand, they show that our trustworthiness approach provides a sound ranking.

Regarding the results for non-vulnerable functions (Table 4.16), again 4 out of the 9 experts indicate the same ranking as our approach, but the order of $Func_A$ and $Func_B$ is reversed in the aggregated ranking. This is happening because 4 experts, that are partially indicating the same order as our approach ($Func_E > Func_D > Func_A > Func_C$), choose $Func_B$ as the least trustworthy function, while it has a higher score than $Func_A$ according to our ranking. For example, Expert6's values for $Func_B$ and $Func_A$ are 0.037 and 0.275, while their scores in our approach using MDAMDG are 0.493 and 0.226, respectively. Thus, despite having several experts with the same rank as ours, this significant distance between our rank and several experts (4 experts) in the case of $Func_B$, greatly influences the result of the aggregated ranking.

To further verify whether the proposed approach for trustworthiness benchmarking is valid, we repeated the whole process with the files and functions containing known vulnerabilities. The results are presented in Tables 4.17 and 4.18 for files and functions, respectively.

Table 4.17: Individual & aggregated rankings of vulnerable files.

Expert 1 Rank		**Expert 2 Rank**		**Expert 3 Rank**		**Expert 4 Rank**		**Expert 5 Rank**	
File**I**	0,302	File**I**	0,570	File**I**	0,281	File**I**	0,379	File**I**	0,531
File**G**	0,302	File**G**	0,177	File**G**	0,281	File**G**	0,244	File**J**	0,186
File**H**	0,302	File**J**	0,177	File**J**	0,281	File**J**	0,142	File**F**	0,142
File**F**	0,064	File**F**	0,037	File**F**	0,106	File**F**	0,142	File**G**	0,101
File**J**	0,029	File**H**	0,037	File**H**	0,050	File**H**	0,093	File**H**	0,041
Expert 6 Rank		**Expert 7 Rank**		**Expert 8 Rank**		**Expert 9 Rank**		Agg. Rank	
File**I**	0,476	File**I**	0,403	File**I**	0,414	File**I**	0,511	File**I**	0,429
File**G**	0,306	File**G**	0,259	File**J**	0,254	File**J**	0,194	File**G**	0,233
File**J**	0,089	File**J**	0,134	File**G**	0,193	File**G**	0,113	File**J**	0,165
File**F**	0,089	File**F**	0,134	File**F**	0,090	File**F**	0,091	File**F**	0,103
File**H**	0,040	File**H**	0,069	File**H**	0,049	File**H**	0,091	File**H**	0,070

Table 4.18: Individual & aggregated rankings of vulnerable functions.

Expert 1 Rank		Expert 2 Rank		Expert 3 Rank		Expert 4 Rank		Expert 5 Rank	
FuncG	0,406	FuncG	0,566	FuncG	0,380	FuncG	0,555	FuncG	0,477
FuncJ	0,406	FuncJ	0,243	FuncJ	0,380	FuncJ	0,230	FuncJ	0,308
FuncF	0,100	FuncF	0,112	FuncF	0,118	FuncF	0,097	FuncF	0,097
FuncH	0,047	FuncH	0,040	FuncH	0,061	FuncH	0,066	FuncH	0,072
FuncI	0,042	FuncI	0,040	FuncI	0,061	FuncI	0,053	FuncI	0,046
Expert 6 Rank		**Expert 7 Rank**		**Expert 8 Rank**		**Expert 9 Rank**		**Agg. Rank**	
FuncG	0,473	FuncG	0,614	FuncG	0,523	FuncG	0,609	FuncG	0,462
FuncJ	0,199	FuncJ	0,136	FuncJ	0,193	FuncJ	0,168	FuncJ	0,272
FuncF	0,160	FuncF	0,136	FuncF	0,174	FuncF	0,108	FuncF	0,153
FuncH	0,093	FuncH	0,057	FuncH	0,055	FuncI	0,070	FuncH	0,057
FuncI	0,075	FuncI	0,057	FuncI	0,055	FuncH	0,045	FuncI	0,055

After analyzing the results, we observed that the files ranking obtained by our trustworthiness approach, $File_I > File_G > File_J > File_F > File_H$, matches the aggregated ranking of the experts, which also maps to the individual ranking of the majority of the experts (5 out of 9). The same happens for the functions. Our ranking is $Func_G > Func_J > Func_F > Func_H > Func_I$, which corresponds to the aggregated ranking and to the individual ranking of the majority of the experts (8 out of 9).

4.4.3 Approach Generalization Discussion

To study the generalization properties of the proposed approach, we repeated the trustworthiness benchmarking process using different software metrics. Since the metrics used for building the trustworthiness model were selected considering the Mozilla Firefox dataset, we need to verify whether the ranking results are biased. In practice, we repeated the whole ranking process for the same files and functions but using another set of software metrics, considered as the best set for distinguishing vulnerable code units from non-vulnerable ones in a dataset containing the combination of all the five projects (Mozilla Firefox, Linux Kernel, Apache, Xen and Glibc).

There are common sofware metrics in the two sets of metrics (set of metrics selected for Mozilla Firefox and set of metrics selected for all projects combined), for both files and functions. For files, the new set of software metrics includes a total of 16 metrics of which 6 are new (AltAvgLineCode, AltCountLineComment, AvgCyclomatic, AvgFanIn, CountLineBlank and SumEssential). For functions, there are also a total of 16 metrics of which 7 are new (AltCountLineCode, CountInput, CountLineCodeDecl, CountPath, CountStmt, Cyclomatic and RatioCommentToCode). Refer to Section 3.2.3 for more details on this subset of metrics.

As shown in tables 4.19, 4.20, 4.21, and 4.22, the same ranking, with slightly different trustworthiness scores, is achieved respectively for files and functions without and with vulnerabilities.

Table 4.19: Rank of files without vulnerabilities.

MDG_W		MDA_W		MDAMDG_W	
Files	**Tscore**	**Files**	**Tscore**	**Files**	**Tscore**
FileC	0,931	**FileC**	0,929	**FileC**	0,930
FileA	0,761	**FileA**	0,677	**FileA**	0,735
FileE	0,527	**FileE**	0,501	**FileE**	0,519
FileD	0,225	**FileD**	0,197	**FileD**	0,216
FileB	0,063	**FileB**	0,059	**FileB**	0,062

Table 4.20: Rank of files with vulnerabilities.

MDG_W		MDA_W		MDAMDG_W	
Files	**Tscore**	**Files**	**Tscore**	**Files**	**Tscore**
FileI	0,957	**FileI**	0,940	**FileI**	0,951
FileG	0,799	**FileG**	0,760	**FileG**	0,787
FileJ	0,603	**FileJ**	0,586	**FileJ**	0,598
FileF	0,250	**FileF**	0,261	**FileF**	0,254
FileH	0,005	**FileH**	0,009	**FileH**	0,006

Table 4.21: Rank of functions without vulnerabilities

MDG_W		MDA_W		MDAMDG_W	
Functions	**Tscore**	**Functions**	**Tscore**	**Functions**	**Tscore**
FuncE	0,906	**FuncE**	0,870	**FuncE**	0,898
FuncD	0,740	**FuncD**	0,712	**FuncD**	0,734
FuncB	0,633	**FuncB**	0,611	**FuncB**	0,628
FuncA	0,555	**FuncA**	0,521	**FuncA**	0,547
FuncC	0,341	**FuncC**	0,318	**FuncC**	0,336

Table 4.22: Rank of functions with vulnerabilities.

MDG_W		MDA_W		MDAMDG_W	
Functions	**Tscore**	**Functions**	**Tscore**	**Functions**	**Tscore**
FuncG	0,910	**FuncG**	0,873	**FuncG**	0,901
FuncJ	0,813	**FuncJ**	0,783	**FuncJ**	0,806
FuncF	0,673	**FuncF**	0,655	**FuncF**	0,669
FuncH	0,481	**FuncH**	0,469	**FuncH**	0,478
FuncI	0,368	**FuncI**	0,337	**FuncI**	0,360

As we can observe, the trustworthiness model-based ranking obtained using the set of software metrics selected for Mozilla for files without vulnerabilities is $File_C > File_A > File_E > File_D > File_B$, the same as the ranking presented in Table 4.19 (obtained using the selected metrics for different projects combined). In addition, the values of the trustworthiness score are similar: $0,916 > 0,723 > 0,482 > 0,245 > 0,051$ (see table 4.8) using the selected metrics for Mozilla project, and $0,930 > 0,735 > 0,519 > 0,216 > 0,062$ (see table 4.19) using the selected metrics for all projects combined. These results suggest that the

approach can be generalized to other software projects, but further experiments are needed to get more confidence in such observation (e.g., considering totally different projects).

4.5 Summary

This chapter presented a framework that aim to move from a subjective to an objective trustworthiness notion by building a model based on several software attributes and their relative importance for determining how trustworthy a piece of software is. The proposed model is based on the architectural quality attributes of software, represented by concrete software metrics of different types, including complexity (e.g., SumCyclomatic), volume (e.g., CountLineCodeDecl), coupling (e.g., CBO), and cohesion metrics (e.g., LCOM). **The main objective is focused on showing the possibility of using software metrics to benchmark the trustworthiness of software systems.**

For validation, we conducted a survey among several software security experts and calculated individual and aggregated ranks of the same files and functions based on their pairwise comparison of files and functions. Those ranks were compared with the ranks obtained with our approach. The results show that the rankings of the experts greatly match the one obtained by our trustworthiness model-based approach, thus, the proposed trustworthiness benchmarking framework is able to provide a sound ranking of the benchmarked files and functions. This demonstrates the possibility of using software metrics for benchmarking the trustworthiness of software systems. We finally showed that the proposed framework can potentially be generalized to other software project.

Despite the results showed a sound ranking of the benchmarked files and functions, **threats to validity** should be highlighted:

- **Number of files and functions selected** - the instantiation and validation of the trustworthiness benchmark were performed using a total of 10 files (5 non-vulnerable and 5 vulnerable) and 10 functions (5 non-vulnerable and 5 vulnerable). Although we obtained the trustworthiness score for all the files and functions of the dataset, we needed to reduce this number in order to perform the validation. This is an acceptable number of files and functions, since the experts had to perform the comparison between all the possible pairs.
- **Number of experts considered** - our aim was to get as many responses as possible from experts in the software security field, however this is a difficult task. Nevertheless, we consider that the results obtained provide a good basis for demonstrating the validity of the proposed approach.

Considering that trustworthiness is an integrating concept composed by several but complementary attributes (e.g., performance, availability, security), this proposal is focused only on security aspects (also to make the problem dealable). However, the proposed framework is applicable, with minimum changes, to any other trustworthiness attribute. It is also worth noting that security is nowadays

a fundamental attribute of trustworthiness in the majority of critical software systems and applications (Mohammadi et al. [2013]).

As we were able to consider trustworthiness as a measurable notion and to propose a **trustworthiness benchmark capable of providing a sound scoring of code units**, in the next chapter we propose solutions to categorize and prioritize such units of codes considering their trustworthiness levels.

Chapter 5

Security Categorization of Code Units

Security is clearly a crucial issue to consider during the design and implementation of any software system. A lot of efforts focused on the definition of best practices, standards, and regulations to help developers in building high quality and secure software have been proposed. However, it is still very difficult for developers, if not impossible, to build software without vulnerabilities. Vulnerabilities, if not uncovered and mitigated during software development, can incur huge cost in terms of time, money and efforts after implementation. Thus, it becomes crucial the avoidance and elimination of vulnerabilities in the early phases of software development process.

To improve the current situation, we need to identify, investigate, and use the early evidences of security issues in the code, in a way that supports developers in the detection of potential issues during the software development process (i.e., design and implementation) (Evans and Larochelle [2002]). To do so, in this chapter we propose a **framework for code categorization based on security evidences**. The idea is to define a mechanism that, based on the early evidences of security issues in the code, supports developers in the identification of potential untrustworthy (insecure) code units starting from the early phases of the coding process.

The mechanism is composed by a set of **characterization models** and a **categorization mechanism**, and can be applied in different **application scenarios** considering different security concerns. It enables a **continuously monitoring of performance** based on the improvement of the security evidences, as the categorization mechanism can be updated using new data/evidences of security gathered over time. The characterization models are built based on Machine Learning prediction models with the aim of classifying code units. Then, the categorization mechanism allows the prioritization of code units for each model taking into account a specific application scenario. Any kind of predic-

tion model, evidences of security issues (e.g. software metrics or code smells) and code units (e.g., files, functions) can be used.

To instantiate the framework, we propose two different characterization models: *i)* **a Consensus-Based Decision-Making (CBDM) approach** capable of grouping code units in several categories based on the *classification result of several prediction models*; and *ii)* **Trustworthiness Models (TMs) to categorize and prioritize code**, not directly based the results of the prediction models, but on the **trustworthiness scores** computed by several trustworthiness models. Both cases are built on top of Machine Learning algorithms and software metrics are used as security evidences.

The outline of this chapter is as follows. First, the framework for code units categorization is described. Then, in Section 5.2, the two instantiations of the framework are presented. Section 5.3 presents the experimental evaluation for the CBDM approach, while the results for the approach based on Trustworthiness Models are in Section 5.4. The chapter closes with a summary of the main insights obtained from the results in Section 5.5.

5.1 Code Units Categorization Framework

Our framework to categorize and prioritize code units using security evidence is intended to help developers to efficiently and effectively review their code, being capable of grouping code units into several categories, representing their trustworthiness level from a security perspective. It does not aim at finding vulnerable code, but instead at raising the attention of developers to the code units that seem to be more problematic.

The process is divided into different stages: *i)* extraction of security evidences, *ii)* definition of characterization models, *iii)* definition of the categorization mechanism, and *iv)* assessment of performance results. The entire process is presented in Figure 5.1 and described below. Note that, a continuous performance monitoring of the mechanism based on the improvement of the security evidences can also be included. In practice, the prediction model can be continuously improved by new data collected from the source code under development, but we leave this for future work.

5.1.1 Extract Security Evidences

Software metrics are widely used indicators of software quality thus, they have been used as security evidence in many studies. However, other evidences rather than software metrics, like **code smells** (Cairo et al. [2018]), **lack of security best practices in the code**, **alerts given by static code analysers**, among others, can be used to improve the detection/prediction models to produce less false alarms and try to find the location and type of vulnerabilities to provide some suggestions to developers for removing the detected or predicted vulnerabilities and improving the code.

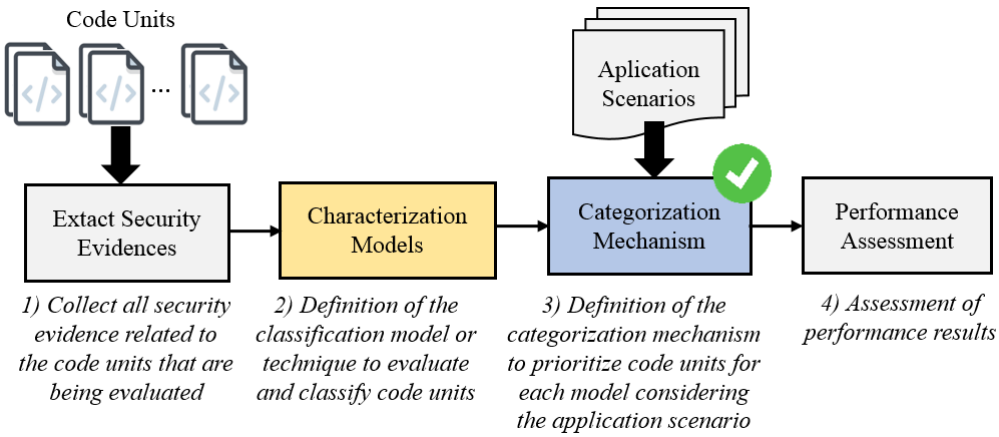


Figure 5.1: Code units categorization methodology.

The goal of this first phase is to collect all the security evidence related to the code units that are being used for building the models. Thus, all data such as software metrics, code smells, security alerts among others, obtained using different tools and techniques, can be used.

5.1.2 Characterization Models

In this phase, the code unit characterization model should be defined. Our goal is for the framework to be flexible in a way that any kind and number of vulnerability prediction models/tools/techniques can be used. Figure 5.2 describes a generic characterization model design process. As shown, the idea is to use different prediction models or any other technique to evaluate and classify code units.

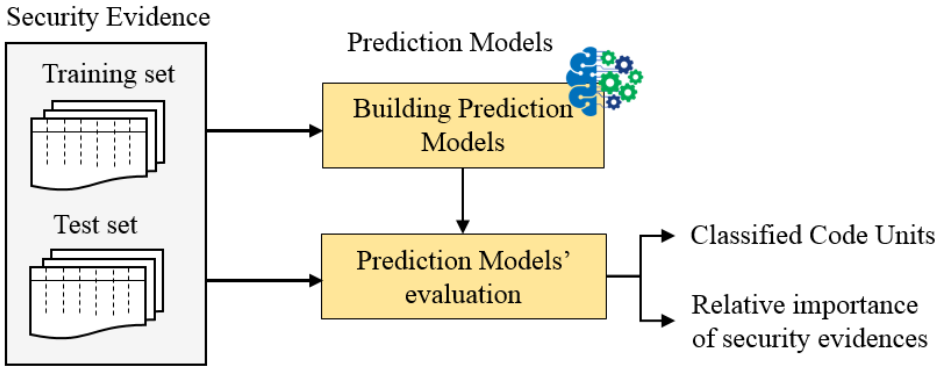


Figure 5.2: Models' characterization.

The input data for this phase are the **security evidence** for each code unit. The supporting dataset should be divided into a *training set* and a *test set*. The trainning set will be used to build the prediction models and the test set serves to evaluate the prediction results. The output of this process are the **classified code units** and the **relative importance of each security evidence** used by the prediction model to classify each unit of code.

The idea is not to define a generic model, but instead to integrate different

models depending on the information to be used during the instantiation of the framework. As mentioned previously, different vulnerability prediction models/tools/techniques can be used as well as different security evidences (e.g. software metrics, code smells). Furthermore, it is possible to assign weights to the prediction models used; however, this is a complex problem, as each prediction model may perform differently in different contexts, and for that reason, in the present work, we consider all the predictors to have the same contribution for the decision.

5.1.3 Categorization Mechanism

Figure 5.3 describes the process proposed for defining the categorization mechanism. Based on the **characterization results** and considering different **application scenarios**, the code units can be grouped into k categories. For example, for each characterization model, the code units can be categorized from *Trustworthy* to *Absolutely Untrustworthy*. To assess and validate the categorization results, a **categorization assessment** should be performed which may consist, for example, on a detailed analysis of the reported vulnerabilities by category.

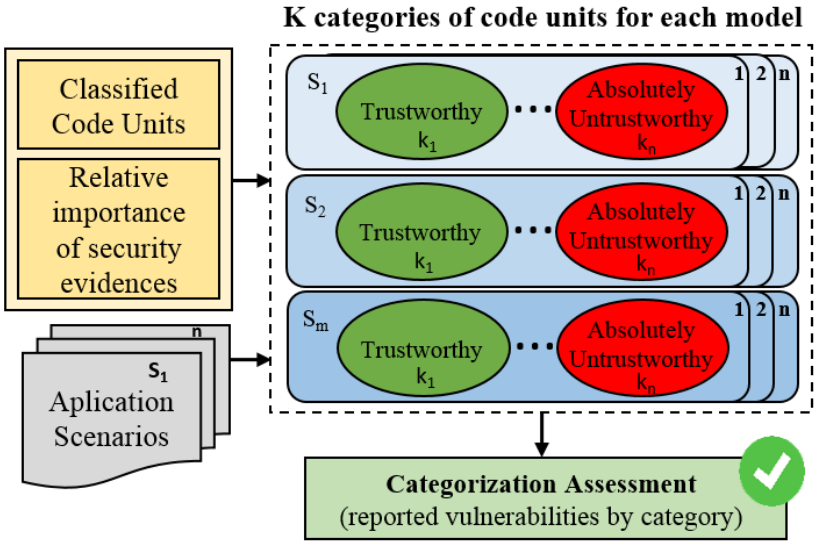


Figure 5.3: Categorization Mechanism.

Our proposal is to categorize the code units into different categories depending on the criticality of the system and the available resources (application scenarios presented in Section 3.3.2). Thus, different thresholds should be defined for each **application scenario**. For example, if a Highly-Critical software system is under evaluation and there are a lot of resources to detect vulnerabilities, the number of code units to be reviewed can be higher. On the other hand, if fewer resources are available, a lower number of code units will be reviewed. This means that increasing the number of categories results in smaller groups with fewer code units. Therefore, when there are fewer resources to review code and eliminate vulnerabilities, it is useful to perform the categorization with a

higher number of clusters (e.g., to better support the decision on which units to consider for being reviewed).

5.1.4 Performance Assessment

To assess the performance of the code classification mechanism (i.e., an instantiation of the proposed framework), we propose two sets of experiments: *i)* to analyze the reported vulnerabilities per category, and *ii)* to perform an expert-based ranking of code units.

For the first case, the idea is to **collected the reported security vulnerabilities** of the project under study from different sources, such as CVEDetails, and then verify the number of code units with reported vulnerabilities in each category (from more Trustworthy to Absolutely Untrustworthy). In other words:

1. Collect the reported security vulnerabilities of the project;
2. Identify the code units that have at least one reported vulnerability;
3. Verify to which category each of these vulnerable code units belong to (i.e., count the number of unique code units with reported vulnerabilities for each category).

For the second set of experiments, we propose the use of **expert-based ranking**. The idea it to randomly select several code units (vulnerable and non-vulnerable) and ask experts to rank them by means of pairwise comparisons (thus the number of code units to be analyzed should be feasible). In practice, the experts should be provided with the source code of each unit under analysis and corresponding value of the security evidences collected (e.g., value of software metrics for each code unit), and asked to perform a pairwise comparison (i.e., assign a value indicating how much trustworthy a code unit is in comparison to another). In order to perform the aggregated rankings of code units based on perceived trustworthiness of the experts, the Aggregation of Individual Judgments (AIJ) method (Dong et al. [2010b]) can be used. The final step is to compare the expert-based ranking with the categorization results provided by our mechanism.

Examples of both experiments to assess the performance of the code classification mechanism will be seen in detail during the framework instantiation results (see sections 5.3 and 5.4).

5.2 Framework Instantiations

Here we present two different instantiations of the framework: first, the Consensus-Based Decision-Making approach and then the Trustworthiness Models to categorize code.

5.2.1 Consensus-Based Decision-Making (CBDM) Approach

The first instance of the proposed framework addresses the hypothesis of developing a Consensus-Based Decision-Making (CBDM) approach on top of several Machine Learning-based prediction models, trained using software metrics data, to categorize code units with respect to their security. The reasoning is that such CBDM solution may be more beneficial for software development teams than the direct output of the vulnerability detection classification models, as discussed in Chapter 3.

In our proposal, depicted in Figure 5.4, the judgment (or decision on whether a code unit is more or less trustworthy) is made by aggregating the classification results from several prediction models that are built using data regarding different security evidences extracted from the code and vulnerabilities previously reported. In concrete, in our instantiation, the prediction models are built by running five different machine learning algorithms (selected in Section 3.3.1) on top of software metrics, to classify code units into two classes: vulnerable or non-vulnerable. The classification results of the five models are then aggregated into a single judgment (or decision), which allows categorizing the assessed code units into five categories that represent different levels of trustworthiness. As mentioned before, different number of categories can be used depending on the criticality of the system and the available resources.

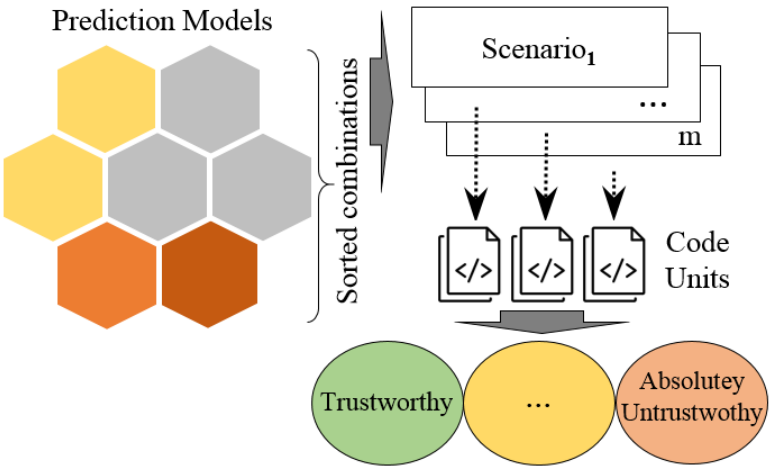


Figure 5.4: Consensus-Based Code Trustworthiness Assessment.

Assigning weights to the prediction models is a complex problem, as each prediction model may perform differently in different contexts. For this reason, we have decided to aggregate the outputs of the prediction models without assigning any weight to each individual model. Thus, we define a model where the combinations of the prediction models are sorted based on the True Positive rate of the classifications. The idea is to compute the number of vulnerable code units that are correctly classified by each Machine Learning model and by all the combinations of the models (we combine the ML results in order to verify if it is possible to increase the number of vulnerable code units classified as vulnerable). The sorted combinations of the prediction models indicate which one is able to correctly classify a greater number of vulnerable code units.

The detailed Consensus-Based Decision-Making (CBDM) approach encompasses the following steps:

1. **Extract security evidences**: software metrics are used as security evidences (28 function level metrics and 51 file level metrics were used, as discussed in Section 3.1);
2. **Build the Prediction Models**: the classification models are built over the dataset of different projects at file and function levels by considering several combinations of software metrics and different application scenarios. As discussed previously in Section 3.3, 75% of the dataset is used to train the Machine Learning models, the prediction models are trained using balanced training sets and an internal and external cross-validation (CV) is performed in all cases. We used five different machine learning algorithms, Random Forest (RF), Decision Tree (DT), Linear and Radial Support Vector Machine, and Extreme Gradient Boosted (Xboost) algorithms.
3. **Evaluation of the Individual Prediction Models**: In all experiments, 25% of the dataset (disjoint from the training sets) is used as a test set in order to compute their classification results.
4. **Combination of the Prediction Results**: Here, we sort all the combinations of the prediction models (2^n combinations for n prediction models) based on the True Positive rate of their combined classification. We give the highest rank to the combination with all prediction models, as the files/functions that are pointed as vulnerable by this combination are absolutely untrustworthy and should be in the highest priority for review by developers. On the other hand, the files/functions that are not classified as potentially vulnerable by any prediction model, are in the lowest rank, which means that they can be considered as trustworthy. Finally, we rank the other combinations based on the number of true positive decisions made (or potentially vulnerable code detected) by the prediction models included in each combination. The combinations with more true positive decisions/classifications have an higher importance in the decision than others. Thus, the code classified by such combinations as vulnerable is less trustworthy, requiring a higher priority for review.
5. **Categorization mechanism**: Based on the results of the prediction models and considering different application scenarios, the code units are grouped into several categories from *trustworthy* to *absolutely untrustworthy*.

Figure 5.5 presents an example of the categorization, considering five different categories:

1. **Absolutely Untrustworthy** (where the files/functions were classified as vulnerable by all machine learning algorithms);
2. **Highly Untrustworthy**;
3. **Untrustworthy**;

4. **Low Trustworthy**;
5. **Trustworthy** (where the files/functions were classified as non-vulnerable by all classifiers).

(a) Categorization of code units

PM_1	$PM_{...}$	PM_n	TP rate	Accum. TP rate	Categories
Comb.$_1$			TPr_1	TPr_1	Trustworthy
Comb.$_2$			TPr_2	$TPr_1 + TPr_2$	Low Trustworthy
Comb.$_3$			TPr_3 ($< TPr_2$)	$TPr_1 + ... + TPr_3$	
...			...	...	Untrustworthy
...			...		
...			...	...	Highly Untrustworthy
Comb.$_{2^{n-1}}$			$TPr_{2^{n-1}}$ ($< TPr_{2^{n-2}}$)	...	
Comb.$_{2^n}$			TPr_{2^n}	100%	Absolutely Untrustworthy

(b) Examples of CBDM set for different scenarios

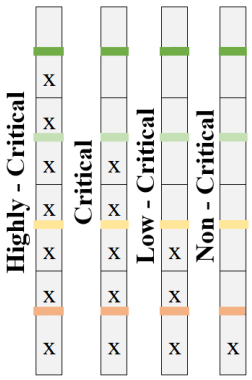


Figure 5.5: Instantiating the Approach with an Example for Different Scenarios.

As we can observe, the number of combinations of prediction models (and consequently the number of code units) included into the **Absolutely Untrustworthy** and the **Trustworthy** categories will be always the same considering different application scenarios. This is related to the fact that these categories include the code units that were classified as vulnerable by all ML algorithms and the code units that were classified as non-vulnerable by all classifiers. The number of combinations of prediction models (and consequently the number of code units) included in the remaining categories (**Highly Untrustwothiness**, **Untrustworthy** and **Low Trusworthy**) will depend on the the thresholds applied. Thus, considering different application scenarios (based on the critically of the system under study and the available resources presented in Section 3.3.2), we can adjust the boundaries of these categories using the True Positive rate. For example, if there are a lot of resources available and we want to review a larger amount of code units, we should define a higher threshold (e.g., TP rate $< 90\%$). This way, a greater number of ML combinations will be included in this category (e.g., Highly Untrustworthy). On the other hand, if a lower TP rate is defined (e.g., TP rate $< 80\%$), less ML combinations will be included and consequently a smaller number of code units to be reviewed. A more detailed example of this adjustment is shown later during the presentation of the instantiation results (see Section 5.3).

5.2.2 Trustworthiness Models (TMs) to Categorize Code

The main idea behind the second instantiation is to use the *trustworthiness score* of the code units (e.g., files/classes or functions/methods) to categorize/prioritize them. To do so, we build on top of the benchmarking approach proposed in Chapter 4 to compute the trustworthiness score of the code units under evaluation. Then, a clustering technique is used to put the code units into several

categories. As before, a category refers to the extent to which a piece of code is prone to be vulnerable (or untrustworthy), which varies from *Absolutely Untrustworthy* to *Highly Trustworthy* (the number of categories applied depends on how much we need to distinguish the code units, as will be discussed later).

As shown in Figure 5.6, each code unit (e.g., function/method, file/class) is assessed by N Trustworthiness Models (TM_1 to TM_N), and is assigned with N scores (each trustworthiness model gives one score to each code unit; S_1 to S_N). After calculating all scores, the units are classified into different categories representing their trustworthiness (or security) level. As mentioned in Chapter 4, the trustworthiness models can be built based on diverse approaches, techniques or tools (in this study, we use several machine learning algorithms to build our TMs) using different evidences of quality or security issues in code (in this work, software metrics are used as evidences to train the machine learning algorithms).

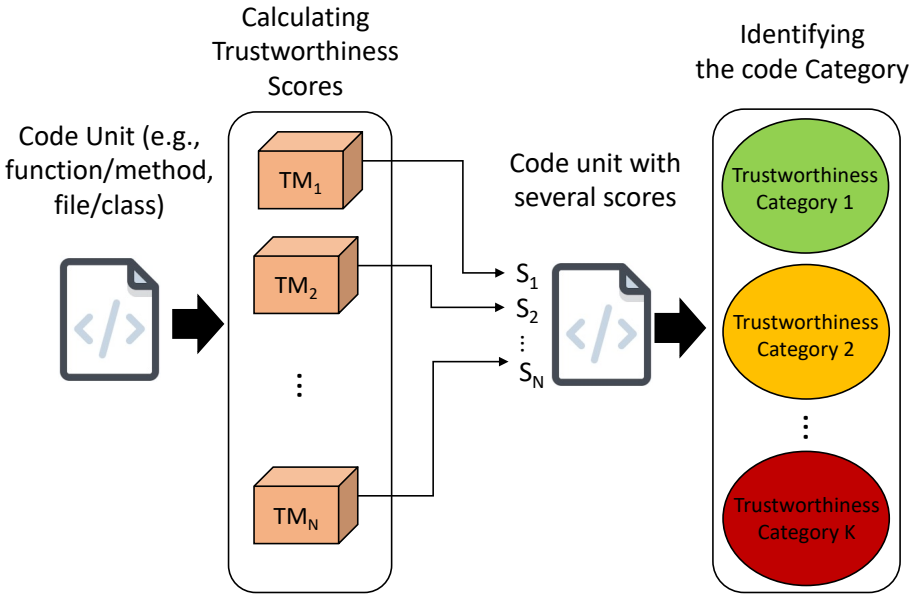


Figure 5.6: Proposed approach for code units categorization.

A more detailed representation of the aforementioned approach is in Figure 5.7, considering five TMs, each one based on a different ML algorithm. The very first step of the process consists in **preparing a dataset** with detailed information about the source code of representative software projects with software metrics and known vulnerabilities (we used the dataset described in Section 3.1).

The second step is to **assign each code unit with several trustworthiness scores**. This requires building the trustworthiness models (TMs) following the *Trustworthiness Assessment* approach presented in Chapter 4, which includes *normalizing the value of software metrics* to a common scale and *computing the relative importance of the software metrics*. To normalize the value of the software metrics to a common scale, we performed a statistical analysis on the input data and then used the **Feature Scaling** process described in Section 4.1.1.

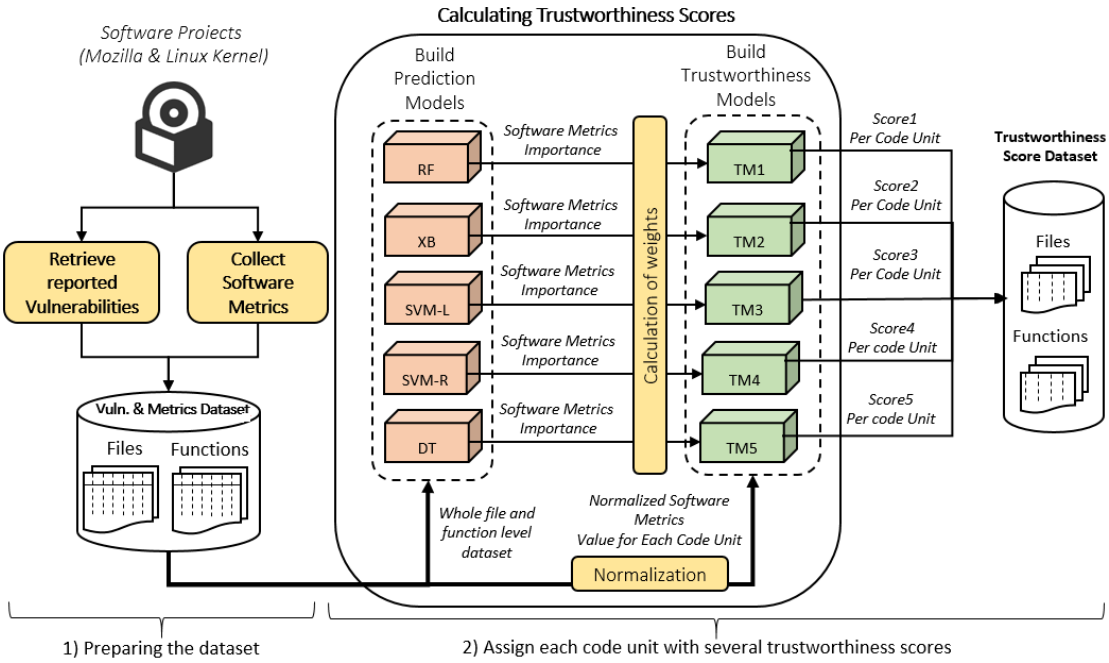


Figure 5.7: Characterization model for approach instantiation.

To compute the weights (i.e. relative importance) of the metrics to feed each TM, we used the importance given to the software metrics by the different Machine Learning algorithms. In practice, a prediction model is trained using a ML algorithm to identify vulnerable code using the software metrics information. The weight given to each software metric by a prediction model indicates its relative importance when used to distinguish vulnerable from non-vulnerable code. In other words, each prediction model reveals how important each software metric is in predicting vulnerable code, and this is used to calculate the software metrics relative weights for each TM. We used the same five machine learning algorithms mentioned before: i) Decision Tree, ii) Random Forest, iii) Linear Support Vector Machine, iv) Radial Support Vector Machine, and v) Extreme Gradient Boosted.

To configure and run the above algorithms, we used the R Project (Team [2017]) and the R Caret package (Kuhn et al. [2020]). The caret package provides a set of functions that streamline the process of creating predictive models. The package contains different functionalities, such as tools for data splitting pre-processing, feature selection and model tuning using resampling variable importance estimation. The features (software metrics in this study) importance was obtained using the *varImp()* function for each classifier.

The five trustworthiness models (TMs) are built based on the **Simple Additive Weighting (SAW)** multi-criteria decision-making method, as discussed in Section 4.1.3).

The last step of the approach (Figure 5.8) is focused on **clustering the code units** to categorize them into different groups representing different trustworthiness levels. Since there is no known set of rules, patterns, or trained models to transform the vector of scores into a trustworthiness category, we applied a

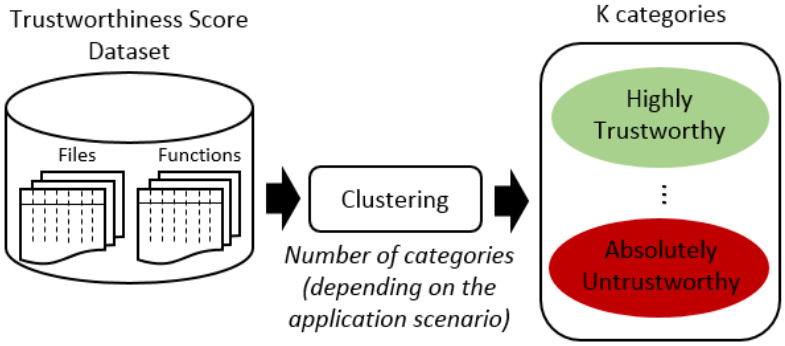


Figure 5.8: Categorization mechanism for approach instantiation.

clustering technique on top of dataset of trustworthiness scores that resulted from the previous step.

Clustering is the task of dividing the data into a certain number of clusters in such a manner that the data belonging to a cluster have similar characteristics (Rai and Singh [2010]). In our case, the clusters are partitioned based on the characteristics of the code units in terms of the trustworthiness score.

There are several techniques to perform clustering, which can be categorized into different types such as, Partitioning techniques, Hierarchical techniques, and Density-based techniques (Saket and Pandya [2016]). The clustering algorithms, in general, follow an iterative process to reassign the data between clusters based on the distance between the clusters (Barioni et al. [2014]). In this instantiation, we use **Partitioning Clustering**, which is one of the most commonly used techniques in the literature.

There are also different types of partitioning clustering methods. The most popular one is the K-means clustering (MacQueen [1967]), in which each cluster is represented by the center or means of the data belonging to the cluster. However, the K-means method is sensitive to outliers. An alternative to K-means clustering is the K-medoids clustering or Partitioning Around Medoids (PAM) (Kaufman and Rousseeuw [1990]), which is less sensitive to outliers compared to K-means. Clustering Large Applications (CLARA) (Gupta and Panda [2019]) is an extension to the PAM algorithm where the computation time has been reduced to make it perform better for large datasets. Since we have large dataset, we decided to use CLARA algorithm.

One important input parameter of clustering algorithms is the number of clusters. In this work, we propose clustering the code units into five clusters corresponding to five different trustworthiness levels (from **Absolutely Untrustworthy** to **Highly Trustworthy**). However, as will be show later, we also considered other values for the number of clusters (3 and 7 clusters) for the sake of comparison. Decreasing and increasing the number of clusters allow us to get larger or smaller groups of code units, respectivelly. This can be applied considering different application scenarios with different resources to review the code.

5.3 CBDM Assessment and Results

In this section, we present and analyze the results obtained for the consensus-based decision-making instantiation of the framework for code categorization. As mentioned, we used five machine learning algorithms on top of software metrics (evidences of potential security issues in the code) to classify the code units into two classes, vulnerable or non-vulnerable. Then, we aggregate the classification results and distribute the code units into five categories representing distinct levels of trustworthiness.

We run the experiments for **files** and **functions** of the Linux Kernel and Mozilla Firefox projects considering the four application scenarios discussed in Section 3.3.2. There are two main reasons to choose these projects: i) both are long duration projects with a large codebase, and ii) both have a considerably high number of reported vulnerabilities (compared to the other three projects in the dataset).

A summary of the projects regarding the number of code units (files and functions) is presented in Table 5.1. The total number of code units used are in the gray columns. The data used for training the predicting models corresponds to 75% of the entire dataset and 25% of it, disjoint from the training sets, was used for testing (presented in green columns). For example, the dataset includes a total of 383622 files for the Linux Kernel project, of which 8712 are vulnerable (i.e., as at least one reported vulnerability); a total of 95905 files were used for testing from which 2178 are labeled as vulnerable and 93727 are labeled as non-vulnerable.

Table 5.1: Summary of the dataset used.

Software Project		# Code units		# Code Units (Test Set)		
		Total	Vuln.	# Vuln.	# Non Vuln.	Total
File-level	**Mozilla Firefox**	185994	3379	844	45591	46435
	Linux Kernel	383622	8712	2178	93727	95905
Function-level	**Mozilla Firefox**	1418482	2780	695	353778	354473
	Linux Kernel	1910776	3021	755	476587	477342

In the next sections, we present the results obtained during the instantiation of the CBDM approach. We start with the the analysis of the best combinations of the prediction models. Then, the categorization based on the application scenarios is presented. Finally, the categorization results are assessed.

5.3.1 Best Combinations of Prediction Models

Table 5.2 shows the best combinations of the machine learning algorithms for the case of **files of the Linux Kernel project**, ordered by the number of correctly classified vulnerable files (i.e., TP rate). In the first five columns, we can see the ML algorithms included in each combination. Then, we have the total number of files and the True Positive Rate per combination. Finally (in the last column), the accumulative value of TPR (sum of the TP rate of the ordered combinations) is presented. For example, we can observe that there are 1047 files (labeled as vulnerable) that are considered vulnerable by all ML algorithms which correspond to 48.07% of TP rate. The **Undetected** group corresponds to the FN (False Negative) files, where we see that a total of 76 files (labeled as vulnerable) are classified as non-vulnerable by all ML algorithms.

	PM_1	PM_2	PM_3	PM_4	PM_5	# Files	TP_{rate}	Accum. TP_{rate}
1	RF	Xboost	SVM_R	SVM_L	DT	1047	48,07%	48,07%
2	RF	Xboost	SVM_R			334	15,34%	63,41%
3	RF	Xboost	SVM_R	SVM_L		207	9,50%	72,91%
4	RF	Xboost	SVM_R		DT	155	7,12%	80,03%
5	RF	Xboost				126	5,79%	85,81%
6	RF	Xboost		SVM_L	DT	87	3,99%	89,81%
7	RF	Xboost			DT	29	1,33%	91,14%
8			SVM_R			25	1,15%	92,29%
9	RF	Xboost		SVM_L		23	1,06%	93,34%
10		Xboost				14	0,64%	93,99%
	Other combinations					55	2,53%	96,51%
	Undetected					76	3,49%	100%

Table 5.2: Best Combinations of Prediction Models for Files of Linux.

Table 5.3 shows the best combinations of the machine learning algorithms for the **functions of Linux Kernel project**. Here, a total of 346 functions are considered vulnerable by all machine learning algorithms, which corresponds to a TP rate of 45.83%. On the other hand, 102 functions are considered non-vulnerable by all ML algorithms (Undetected group). We can see that the best combinations (with higher TP rate) change when analyzing files and functions within the same project. However, there are some combinations that remain the best even when different code levels are analyzed, for example RF- Xboost-SVM_R, RF-Xboost-SVM_R-SVM_L, and RF-Xboost-SVM_R-DT. It is also verified that the best combinations include mostly the prediction models that individually have better performances (RF and Xboost), which was the expected (refer to the performance of the Machine Learning algorithms presented in Section 3.3.4.1).

The results of the best combinations for the **Mozilla Firefox project** are presented in tables 5.4 and 5.5 for **files** and **functions**, respectively. A total of 358 files are considered vulnerable by all machine learning algorithms, which corresponds to a TP rate of 42.42% and 72 files are considered non-vulnerable

	PM_1	PM_2	PM_3	PM_4	PM_5	# Functions	TP_{rate}	Accum. TP_{rate}
1	RF	Xboost	SVM_R	SVM_L	DT	346	45,83%	45,83%
2	RF	Xboost	SVM_R			72	9,54%	55,36%
3	RF	Xboost				66	8,74%	64,11%
4	RF	Xboost	SVM_R	SVM_L		43	5,70%	69,80%
5	RF	Xboost	SVM_R		DT	34	4,50%	74,30%
6			SVM_R			16	2,12%	76,42%
7	RF	Xboost		SVM_L	DT	10	1,32%	77,75%
8		Xboost				9	1,19%	78,94%
9	RF					7	0,93%	79,87%
10	RF		SVM_R			6	0,79%	80,66%
	Other combinations					44	5,83%	86,49%
	Undetected					102	13,51%	100%

Table 5.3: Best Combinations of Prediction Models for Functions of Linux.

by all machine learning algorithms (Undetected group). For functions, 366 are considered vulnerable by all ML algorithms (TP rate of 52.7%) and 67 functions are considered non-vulnerable by all ML algorithms. As for Linux Kernel, the best combinations (with higher TP rates) change when comparing the results of files and functions.

	PM_1	PM_2	PM_3	PM_4	PM_5	# Files	TP_{rate}	Accum. TP_{rate}
1	RF	Xboost	SVM_R	SVM_L	DT	358	42,42%	42,42%
2	RF	Xboost	SVM_R			100	11,85%	54,27%
3	RF	Xboost	SVM_R		DT	50	5,92%	60,19%
4	RF	Xboost		SVM_L	DT	46	5,45%	65,64%
5	RF	Xboost	SVM_R	SVM_L		34	4,03%	69,67%
6	RF	Xboost				28	3,32%	72,99%
7			SVM_R			24	2,84%	75,83%
8				SVM_L	DT	22	2,61%	78,44%
9	RF	Xboost			DT	15	1,78%	80,21%
10		Xboost				15	1,78%	81,99%
	Other combinations					80	9,48%	91,47%
	Undetected					72	8,53%	100%

Table 5.4: Best Combinations of Prediction Models for Files of Mozilla.

When comparing the results between the two projects, it is interesting to verify that the best combinations in case of files, presented in tables 5.2 and 5.4, are very similar. In fact, nine out of the ten best combinations are the same on both projects, but the same is not true for functions. This probably happens due to the imbalanced nature of the dataset (number of vulnerable and non-vulnerable records presented in Table 5.1). In fact the function-level data is even more imbalanced than the file-level, thus affecting the performance of the machine learning algorithms (as mentioned before in Section 3.3.4.1). Also, it is important to mention that each prediction model or combination of prediction models

	PM_1	PM_2	PM_3	PM_4	PM_5	# Functions	TP_{rate}	Accum. TP_{rate}
1	RF	Xboost	SVM_R	SVM_L	DT	366	52,7%	52,66%
2	RF	Xboost				39	5,6%	58,27%
3					DT	33	4,7%	63,02%
4	RF	Xboost	SVM_R		DT	30	4,3%	67,34%
5	RF	Xboost			DT	22	3,2%	70,50%
6	RF	Xboost	SVM_R	SVM_L		19	2,7%	73,24%
7	RF		SVM_R	SVM_L	DT	17	2,4%	75,68%
8			SVM_R	SVM_L	DT	16	2,3%	77,99%
9	RF	Xboost	SVM_R			14	2,0%	80,00%
10				SVM_L	DT	9	1,3%	81,29%
	Other combinations					63	9,1%	90,36%
	Undetected					67	9,64%	100%

Table 5.5: Best Combinations of Prediction Models for Functions of Mozilla.

perform differently in different contexts and using the two code levels (files and functions) as input data. Thus, the best combinations must be identified for each project and for code level.

5.3.2 Categorization based on the Application Scenarios

As explained before, the files that are classified as vulnerable by all 5 classifiers go to the absolutely untrustworthy category (regardless of being TPs or FPs, considering the labels in the original dataset). In contrast, files classified as non-vulnerable (TNs and FNs) by all 5 classifiers go to the trustworthy category. Regarding the intermediate categories we need to define the thresholds considering the four application scenarios. These thresholds are based on the accumulative true positive rate in each scenario. Table 5.6 presents an example of the files categorization for the Linux Kernel project considering Highly-Critical, Critical, Low-Critical and Non-Critical scenarios.

For the Highly-Critical scenario, the intermediate categories relate to a TP rate higher than 95%, between 90% and 95% and less that 90% (Table 5.6 (a)). In this example, the limit of the *low trustworthy* category is set on the combination at which the cumulative true positive rate exceeds 95%. This means that this category includes all files minus the *trustworthy* files and minus the files that are detected by the combinations that could detect 95% or less of the vulnerable files during training. Similarly, the *untrustworthy* category includes all files minus the *trustworthy* files, the *low trustworthy* files, and the files that are detected as vulnerable by the combinations that could detect 90% or less of the vulnerable files. Finally, the *highly untrustworthy* category includes all the files minus the ones included in the previous categories and the *absolutely untrustworthy* ones.

Similar definitions are presented for the other scenarios by changing the value of thresholds for each category (Table 5.6 (b), (c), and (d)). By analyzing the number of files in each category in each scenario in Table 5.6, we can see that

Table 5.6: Linux Kernel file categorization based on the scenarios.

(a) Highly - Critical

Categories	Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
Trustworthy	FN (all)	76	61803	61879	64,5%
Low Trust.	>95%	33	13550	13583	**14,2%**
Untrust.	90 - 95%	84	12084	12168	**12,7%**
Highly Untrust.	< 90%	938	3349	4287	**4,5%**
Abs. Untrust.	TP (all)	1047	2941	3988	4,2%
	Total	**2178**	**93727**	**95905**	

(b) Critical

Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
FN (all)	76	61803	61879	64,5%
>90%	117	25634	25751	**26,9%**
85 - 90%	116	1310	1426	**1,5%**
< 85%	822	2039	2861	**3,0%**
TP (all)	1047	2941	3988	4,2%
Total	**2178**	**93727**	**95905**	

(c) Low - Critical

Categories	Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
Trustworthy	FN (all)	76	61803	61879	64,5%
Low Trust.	>85%	233	26944	27177	**28,3%**
Untrust.	80 - 85%	126	401	527	**0,5%**
Highly Untrust.	< 80%	696	1638	2334	**2,4%**
Abs. Untrust.	TP (all)	1047	2941	3988	4,2%
	Total	**2178**	**93727**	**95905**	

(d) Non - Critical

Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
FN (all)	76	61803	61879	64,5%
>85%	359	27345	27704	**28,9%**
70 - 80%	155	370	525	**0,5%**
< 70%	541	1268	1809	**1,9%**
TP (all)	1047	2941	3988	4,2%
Total	**2178**	**93727**	**95905**	

Table 5.7: Linux Kernel function categorization based on the scenarios.

(a) Highly - Critical

Categories	Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
Trustworthy	FN (all)	102	282676	282778	59,2%
Low Trust.	>85%	12	26803	26815	**5,6%**
Untrust.	80 - 85%	38	46392	46430	**9,7%**
Highly Untrust.	< 80%	257	86880	87137	**18,3%**
Abs. Untrust.	TP (all)	346	33836	34182	7,2%
	Total	**755**	**476587**	**477342**	

(b) Critical

Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
FN (all)	102	282676	282778	59,2%
>80%	50	73195	73245	**15,3%**
75 - 80%	42	47486	47528	**10,0%**
< 75%	215	39394	39609	**8,3%**
TP (all)	346	33836	34182	7,2%
Total	**755**	**476587**	**477342**	

(c) Low - Critical

Categories	Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
Trustworthy	FN (all)	102	282676	282778	59,2%
Low Trust.	>75%	92	120681	120773	**25,3%**
Untrust.	70 - 75%	34	9088	9122	**1,9%**
Highly Untrust.	< 70%	181	30306	30487	**6,4%**
Abs. Untrust.	TP (all)	346	33836	34182	7,2%
	Total	**755**	**476587**	**477342**	

(d) Non - Critical

Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
FN (all)	102	282676	282778	59,2%
>70%	126	129769	129895	**27,2%**
65 - 70%	43	6295	6338	**1,3%**
< 65%	138	24011	24149	**5,1%**
TP (all)	346	33836	34182	7,2%
Total	**755**	**476587**	**477342**	

the distribution of files changes from one scenario to another. For example, for the Highly-Critical scenario there are 33, 84 and 938 files categorized as *low trustworthy*, *untrustworthy* and *highly untrustworthy*, respectively. On the other hand, for the Non-Critical scenario there are 359, 155 and 541 files categorized as *low trustworthy*, *untrustworthy* and *highly untrustworthy*, respectively. Note that, when dealing with highly critical systems, we expect to have more files categorized as *highly untrustworthy*. In contrast, when dealing with non-critical systems, we expect to have fewer files in this category. Thus, the results show that in fact there are more files categorized as *highly untrustworthy* in Highly-

Critical scenario than in Non-Critical scenario.

Table 5.8: Mozilla Firefox file categorization based on the scenarios.

(a) Highly - Critical

Categories	Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
Trustworthy	FN (all)	72	25713	25785	55,5%
Low Trust.	>85%	47	5428	5475	**11,8%**
Untrust.	80 - 85%	48	2740	2788	**6,0%**
Highly Untrust.	< 80%	319	9178	9497	**20,5%**
Abs. Untrust.	TP (all)	358	2532	2890	6,2%
	Total	**844**	**45591**	**46435**	

(b) Critical

Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
FN (all)	72	25713	25785	55,5%
>80%	95	8168	8263	**17,8%**
75 - 80%	37	2539	2576	**5,5%**
< 75%	282	6639	6921	**14,9%**
TP (all)	358	2532	2890	6,2%
Total	**844**	**45591**	**46435**	

(c) Low - Critical

Categories	Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
Trustworthy	FN (all)	72	25713	25785	55,5%
Low Trust.	>75%	132	10707	10839	**23,3%**
Untrust.	70 - 75%	52	3602	3654	**7,9%**
Highly Untrust.	< 70%	230	3037	3267	**7,0%**
Abs. Untrust.	TP (all)	358	2532	2890	6,2%
	Total	**844**	**45591**	**46435**	

(d) Non - Critical

Accum. TPrate	# Vuln. Files	# Non. vuln. Files	Total per category	% per cat.
FN (all)	72	25713	25785	55,5%
>75%	184	14309	14493	**31,2%**
65 - 70%	34	300	334	**0,7%**
< 65%	196	2737	2933	**6,3%**
TP (all)	358	2532	2890	6,2%
Total	**844**	**45591**	**46435**	

Similar observations can be made by analyzing the results of functions for Linux Kernel presented in Table 5.7 and for files and functions of the Mozilla Firefox project presented in tables 5.8 and 5.9, respectively. The percentage of files/functions categorized as *Highly Untrustworthiness* and *Low Trustworthy* increase when we compare Non-Critical to Highly-Critical scenarios. As development teams are expected to spend more resources to detect vulnerabilities in Critical scenarios, then more code units can be selected to be reviewed.

Table 5.9: Mozilla Firefox function categorization based on the scenarios.

(a) Highly - Critical

Categories	Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
Trustworthy	FN (all)	67	155985	156052	44,0%
Low Trust.	>85%	40	36113	36153	**10,2%**
Untrust.	80 - 85%	46	25696	25742	**7,3%**
Highly Untrust.	< 80%	176	80089	80265	**22,6%**
Abs. Untrust.	TP (all)	366	55895	56261	15,9%
	Total	**695**	**353778**	**354473**	

(b) Critical

Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
FN (all)	67	155985	156052	44,0%
>80%	86	61809	61895	**17,5%**
75 - 80%	33	18396	18429	**5,2%**
< 75%	143	61693	61836	**17,4%**
TP (all)	366	55895	56261	15,9%
Total	**695**	**353778**	**354473**	

(c) Low - Critical

Categories	Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
Trustworthy	FN (all)	67	155985	156052	44,0%
Low Trust.	>75%	119	80205	80324	**22,7%**
Untrust.	70 - 75%	41	11615	11656	**3,3%**
Highly Untrust.	< 70%	102	50078	50180	**14,2%**
Abs. Untrust.	TP (all)	366	55895	56261	15,9%
	Total	**695**	**353778**	**354473**	

(d) Non - Critical

Accum. TPrate	# Vuln. Func.	# Non. vuln. Func.	Total per category	% per cat.
FN (all)	67	155985	156052	44,0%
>70%	160	91820	91980	**25,9%**
65 - 70%	30	9439	9469	**2,7%**
< 65%	72	40639	40711	**11,5%**
TP (all)	366	55895	56261	15,9%
Total	**695**	**353778**	**354473**	

5.3.3 Assessment of the Categorization Results

To evaluate our approach, we collected (from CVEDetails) the vulnerabilities reported on the **Linux Kernel** after 2017 (remember that the dataset includes only vulnerabilities reported until 2016). Based on these reports, we were able to identify that new vulnerabilities affected a total of 801 files and 650 functions. It is important to mention that, since these are new vulnerabilities, they were not included in the training set used to train the machine learning algorithms. To perform the assessment of the categorization results, we verified in which cluster those 801 files and 650 functions were included(i.e., for each cluster, we counted the number of unique files and functions with new reported vulnerabilities).

Table 5.10 presents the assessment results considering the categorization for the Highly-Critical scenario of Linux Kernel files (see also Table 5.6 (a)).

In practice, the table presents the categorization of the files that had vulnerabilities before 2017 (i.e. labeled as vulnerable in the dataset) and the number of those files that again had at least one vulnerability reported in 2017 or after. For example, among the 33 files that were categorized in the *Low Trustworthy* category, 4 had at least one reported vulnerability in 2017 or afterwards, and from the 84 categorized as *Untrustworthy*, 7 had at least one reported vulnerability. Table 5.10 (b) shows the same type of information but for files that were never reported has vulnerable before 2017 (i.e. labeled as non-vulnerable in the dataset).

Table 5.10: Reported Vulnerabilities in files of Linux Kernel (Since 2017).

	(a) Vulnerable files (dataset)			(b) Non-vulnerable files (dataset)		
Categories	**# Files**	**# Files with vuln. (after 2017)**	**% Files with vuln. (after**	**# Files**	**# Files with vuln. (after 2017)**	**% Files with vuln. (after 2017)**
Trustworthy	76	4	5,3%	61803	672	1,1%
Low Trust.	33	4	12,1%	13550	527	3,9%
Untrustworthy	84	7	8,3%	12084	585	4,8%
Highly Untrust.	938	153	16,3%	3349	302	9,0%
Abs. Untrust.	1047	382	36,5%	2941	487	16,6%
Total	**2178**	550	-	**93727**	2573	-

As we can see, the percentage of vulnerable files increases from the *trustworthy* category to the *absolutely untrustworthy* category. This shows that, in fact, the files categorized as *absolutely untrustworthy* are more prone to be problematic. Considering the files labeled as vulnerable in the original dataset, we verified that, from the ones classified in the *absolutely untrustworthy* category, 36,5% were reported as vulnerable in 2017 or after. On the other hand, only 5,3% of the files in the *trustworthy* category were reported as vulnerable. The same pattern can be observed for the files labeled as non-vulnerable in the original dataset.

An interesting observation is that, 2573 out of the 93727 files (2.7%) that are labeled as non-vulnerable in our dataset had exploitable vulnerabilities. The

ratio is much higher for the files in the absolutely *untrustworthy category*, for which 16.6% of non-vulnerable files were later discovered to be exploitable. We also observe that 550 of 2178 files labeled as vulnerable in the dataset are reported as vulnerable after 2017, among which 382 and 153 are categorized as *absolutely untrustworthy* and *highly untrustworthy*, respectively.

We repeated this evaluation process for the functions of the Linux Kernel project. The results are presented in Table 5.11. As we can verify, although we observed a slight increase in the the percentage of vulnerable functions from the *trustworthy* category to the *absolutely untrustworthy* category, this increase is not so significant comparing with the files results. This probably happens due to the fact that the number of collected vulnerabilities (corresponding to a total of 650 functions) is very low considering the total number of functions (477342), which limits this analysis.

Table 5.11: Reported Vulnerabilities in functions of Linux Kernel (Since 2017).

	(a) Vulnerable functions (dataset)			(b) Non-vulnerable functions (dataset)		
Categories	**# Functions**	**# Functions with vuln. (after 2017)**	**% Functions with vuln. (after 2017)**	**# Functions**	**# Functions with vuln. (after 2017)**	**% Functions with vuln. (after 2017)**
Trustworthy	102	22	21,6%	282676	6846	2,4%
Low Trust.	12	3	25,0%	26803	925	3,5%
Untrustworthy	38	7	18,4%	46392	1292	2,8%
Highly Untrust.	257	64	24,9%	86880	2888	3,3%
Abs. Untrust.	346	119	34,4%	33836	1939	5,7%
Total	**755**	215	-	**476587**	13890	-

For the **Mozilla Firefox** project we had to follow a different approach. Although we also collected the list of security vulnerabilities (from CVEDetails) reported after 2017, the corresponding vulnerable files and functions were not identified in most cases (i.e., we could not get the information needed to identify the files and functions that were affected by these vulnerabilities). For this reason, we were not able to use this information, and instead conducted an expert-based ranking based on the perceived trustworthiness.

To facilitate the work, we used the results of the expert-based ranking presented in Section 4.4, were the Aggregation of Individual Judgments (AIJ) method was used to obtain an aggregated score for 10 files and 10 functions. Tables 5.12 and 5.13 presents the categorization results (Category column) and the expert-based ranking (Agregated Rank of Experts column) for the 10 files and 10 functions, respectively. Both of tables are ordered by the aggregated rank of the experts, from the more trustworthy to the less trustworthy file/function.

As we can observe, most of the files that were considered more trustworthy by the experts were also categorized as trustworthy by our approach and the files considered less trustworthy by the experts were categorized as *Low Trustworthy* or *Highly untrustworthy*. An interesting case is file 45301201 that was considered *Untrustworthy* by the experts and categorized as *Highly untrustworthy* by our approach. This is a file that we know has a vulnerability (column *Affected* in the

table). On the other hand, the two files that were considered more untrustworthy by the experts, were categorized as *Low Trustworthy*, and no vulnerabilities are known so far for those two files.

The ranking results for the 10 functions of the Mozilla Firefox, presented in Table 5.13, show that the our categorization mostly matches the ranking by the experts, again suggesting that our approach is able to achieve a meaningful categorization result.

Table 5.12: Expert-based ranking of files of Mozilla Firefox.

ID File	Affected	Category	Aggregated Rank of Experts
60426601	v	Trustworthy	0,429
263601	nv	Trustworthy	0,408
1701201	nv	Trustworthy	0,31
51355601	v	Trustworthy	0,233
1418301	nv	Trustworthy	0,171
52292101	v	Low trustworthy	0,165
2201801	v	Low trustworthy	0,103
45301201	v	Highly untrustworthy	0,07
22256501	nv	Low trustworthy	0,065
19960501	nv	Low trustworthy	0,046

Table 5.13: Expert-based ranking of functions of Mozilla Firefox.

ID Function	Affected	Category	Aggregated Rank of Experts
361442101	v	Trustworthy	0,462
20082901	nv	Trustworthy	0,458
121633001	nv	Trustworthy	0,303
304227601	v	Trustworthy	0,272
311237201	v	Untrustworthy	0,153
119075401	nv	Highly untrustworthy	0,094
113275001	nv	Trustworthy	0,082
53225001	nv	Abs. untrustworthy	0,063
314376601	v	Highly untrustworthy	0,057
328555501	v	Highly untrustworthy	0,055

5.4 TMs Assessment and Results

For the second instantiation, we used again the Linux Kernel (kernel.org) and Mozilla Firefox (mozilla.org) projects. Table 5.14 presents the summary of the data for the two projects. The Linux Kernel dataset includes 95905 files, of which 2178 have at least one known vulnerability, and 477342 functions, of which 755 are vulnerable. The Mozilla Firefox dataset consists of 46444 files and 354480 functions, of which 844 and 698 are vulnerable, respectively. The number of files and functions of Mozilla Firefox project is slightly different from the dataset presented in Section 5.3. This is due to the fact that, in order to asses the categorization results of Mozilla Firefox project we resort to the expert-based ranking performed in Section 4.4. Thus, the 10 files and 10 functions had

to be included in the test set. Since the test set used corresponds to 25% of the dataset and is randomly chosen, it was necessary to include some of these files manually.

Table 5.14: Summary of the dataset used.

Software Project	File-level			Function-level		
	# Vuln.	# Non Vuln.	Total	# Vuln.	# Non Vuln.	Total
Linux Kernel	2178	93727	95905	755	476587	477342
Mozilla Firefox	844	45600	46444	698	353782	354480

As before, the dataset includes a total of 51 file-level metrics and 28 function-level metrics (previously presented). The complete list of software metrics and a short description of each is presented in tables 3.1 and 3.2 in Chapter 3. The whole dataset, details of all the analysis performed, and the obtained results are available online[1].

In the following sections, we discuss the building of the Trustworthiness Models. Then, we present and analyse the clustering results. Finally, we present a categorization experiment considering different number of clusters.

5.4.1 Building the Trustworthiness Models

To build the trustworthiness models, we started by computing the **relative importance (weight) of each software metric** considering the five machine learning algorithms at hands (Random Forest, Decision Tree, Extreme Gradient Boosted, and Linear and Radial Support Vector Machine), for both file and function level metrics of the Linux Kernel and Mozilla Firefox projects. The relative importance was then normalized into the [0,1] range.

The results are presented in tables 5.15 and 5.16 for file-level and function-level, respectively. All 51 file-level metrics and 28 function-level metrics are listed. The results show that the importance of metrics might be different in different prediction models. The metrics calculated using RF, DT, Xboost, and SVM are different from each other; however, they are the same in the case of Linear and Radial SVM (thus, they are presented in a single column (SVM)). As we can see, the relative importance of the software metrics also varies from one level to another. These observations strongly support the idea behind our proposed approach: we need an integration of several prediction models to make a more precise decision about a piece of code.

It is worth noting that there are tools like LIME (Ribeiro et al. [2016]) that can help to analyze and interpret the results of the prediction models in a more precise way. However, in this work, the prediction models are used as an example for building trustworthiness scoring models (to obtain the weight of software

[1]The dataset used in this study, the obtained results and detailed analysis of the results are available at `https://github.com/nadiapsm/Access-2022`

metrics). For this reason, a detailed analysis of the results of the prediction models is out of the scope.

In the next step, the value of the **software metrics were normalized** into a common range (between 0 and 1). Table 5.17 presents examples of the real and normalized values for 4 (out of 51) file-level metrics of five files of which three have known vulnerabilities (labeled with **v**), and two do not have known vulnerabilities (labeled with **nv**), as we can see in *Label* column.

When the normalized value is 0, that means that the original value is higher than the upper fence and is considered an outlier (e.g., the value of *HK* for files 2, 4, and 5). On the other hand, when the normalized value is 1, it means that the original value is always 0 (e.g., the value of *CountLineInactive* for file 2). For the remaining cases, the Equation 4.2 presented in Chapter 4 is used.

By **using the trustworthiness models**, we obtained four scores for each file and function of each project (the weights of Linear and Radial SVM are equal). An example of the results using files of the Linux Kernel project is presented in Table 5.18. As a higher score represents a more trustworthy code unit, the examples are ordered from more trustworthy to less trustworthy in the table.

Interestingly, the order of the files is the same considering the different scores. Only the third file (File ID: 15866802) in the case of RF has a slightly higher score than the second file. However, the scores given to each file have different values. Based on this simple example, we might be able to rank the files based on their combined scores (for instance, by calculating the average value), but that is not easy/possible when the number of files increases and the files do not maintain the same order for all scores.

5.4.2 Clustering Results and Discussion

The output of the trustworthiness models (i.e., scores) is used as input for the clustering process. For each file and function of both projects, clustering was performed five times: four times based on the individual scores of each model (scores calculated by TMs built using linear SVM and Radial SVM are the same), and one time based on the combination of all scores. The number of clusters in each run is set to 5. The results are presented and discussed in the following subsections. We start by presenting the cluster results of Linux Kernel and Mozilla Firefox projects and then discuss the assessment of the results.

5.4.2.1 Clusters of Linux Kernel and Mozilla Firefox projects

Table 5.19 presents the results obtained for Linux Kernel files (a) and functions (b). Therefore, Table 5.20 presents the results obtained for Mozilla Firefox files (a) and functions (b).

The first four columns (i.e., Score RF, Score DT, Score Xboost, and Score SVM) show the results of the clustering performed on each individual score and the last column shows the result of the clustering performed on the combination of all scores.

Table 5.15: File-level metrics Weight.

	(a) Linux Kernel				(b) Mozilla Firefox			
Software Metrics	**RF**	**DT**	**Xboost**	**SVM**	**RF**	**DT**	**Xboost**	**SVM**
SumEssential	0,032	0,215	0,167	0,059	0,022	0,200	0,095	0,052
MaxEssential	0,030	0,219	0,120	0,058	0,022	0,000	0,000	0,045
SumCyclomaticStrict	0,023	0,181	0,065	0,054	0,020	0,000	0,000	0,051
CountStmtExe	0,021	0,168	0,029	0,053	0,018	0,000	0,000	0,049
SumCyclomatic	0,018	0,169	0,000	0,053	0,018	0,000	0,000	0,050
CountLineCodeExe	0,022	0,000	0,054	0,048	0,017	0,000	0,026	0,050
AltCountLineComment	0,025	0,000	0,051	0,044	0,021	0,200	0,058	0,050
CountLineCode	0,022	0,000	0,040	0,048	0,020	0,000	0,052	0,051
AltCountLineBlank	0,022	0,000	0,037	0,047	0,020	0,201	0,091	0,052
CountLineBlank	0,020	0,000	0,038	0,046	0,018	0,195	0,000	0,051
AvgEssential	0,024	0,027	0,000	0,049	0,015	0,000	0,000	0,000
CountLine	0,020	0,000	0,029	0,050	0,019	0,000	0,000	0,052
MaxCyclomaticModified	0,021	0,000	0,031	0,046	0,021	0,000	0,000	0,000
CountStmt	0,017	0,000	0,028	0,051	0,015	0,000	0,000	0,050
CountLinePreprocessor	0,022	0,000	0,061	0,000	0,017	0,000	0,044	0,048
MaxCyclomaticStrict	0,021	0,011	0,000	0,050	0,022	0,000	0,030	0,000
AltCountLineCode	0,022	0,000	0,000	0,050	0,018	0,000	0,029	0,052
SumCyclomaticModified	0,019	0,000	0,000	0,052	0,021	0,000	0,031	0,051
CountDeclFunction	0,026	0,000	0,000	0,044	0,020	0,000	0,030	0,047
CountLineInactive	0,024	0,000	0,045	0,000	0,027	0,000	0,079	0,000
CountSemicolon	0,019	0,000	0,000	0,050	0,014	0,000	0,031	0,050
CountLineComment	0,027	0,000	0,041	0,000	0,024	0,204	0,134	0,050
MaxCyclomatic	0,018	0,000	0,000	0,047	0,020	0,000	0,000	0,000
CountLineCodeDecl	0,026	0,000	0,040	0,000	0,016	0,000	0,026	0,051
RatioCommentToCode	0,026	0,000	0,033	0,000	0,028	0,000	0,047	0,000
CountStmtDecl	0,019	0,000	0,039	0,000	0,020	0,000	0,055	0,049
AvgLine	0,025	0,000	0,026	0,000	0,024	0,000	0,000	0,000
CountStmtEmpty	0,021	0,000	0,027	0,000	0,025	0,000	0,000	0,000
AvgCyclomaticStrict	0,016	0,010	0,000	0,000	0,015	0,000	0,000	0,000
AvgLineCode	0,024	0,000	0,000	0,000	0,023	0,000	0,028	0,000
MaxFanIn	0,023	0,000	0,000	0,000	0,019	0,000	0,000	0,000
AltAvgLineCode	0,023	0,000	0,000	0,000	0,024	0,000	0,000	0,000
AvgFanIn	0,022	0,000	0,000	0,000	0,014	0,000	0,000	0,000
MaxFanOut	0,020	0,000	0,000	0,000	0,020	0,000	0,000	0,000
CountPath	0,019	0,000	0,000	0,000	0,024	0,000	0,000	0,000
AltAvgLineComment	0,019	0,000	0,000	0,000	0,016	0,000	0,000	0,000
AvgLineBlank	0,018	0,000	0,000	0,000	0,013	0,000	0,000	0,000
AvgLineComment	0,018	0,000	0,000	0,000	0,017	0,000	0,000	0,000
HK	0,018	0,000	0,000	0,000	0,020	0,000	0,000	0,000
AltAvgLineBlank	0,018	0,000	0,000	0,000	0,013	0,000	0,000	0,000
MaxNesting	0,017	0,000	0,000	0,000	0,018	0,000	0,000	0,000
FanIn	0,016	0,000	0,000	0,000	0,023	0,000	0,030	0,000
FanOut	0,014	0,000	0,000	0,000	0,024	0,000	0,058	0,000
AvgFanOut	0,013	0,000	0,000	0,000	0,020	0,000	0,000	0,000
AvgMaxNesting	0,013	0,000	0,000	0,000	0,017	0,000	0,000	0,000
SumMaxNesting	0,013	0,000	0,000	0,000	0,017	0,000	0,000	0,000
AvgCyclomaticModified	0,012	0,000	0,000	0,000	0,013	0,000	0,000	0,000
AvgCyclomatic	0,011	0,000	0,000	0,000	0,013	0,000	0,000	0,000
MaxMaxNesting	0,011	0,000	0,000	0,000	0,014	0,000	0,000	0,000
CBO	0,010	0,000	0,000	0,000	0,034	0,000	0,000	0,000
LCOM	0,000	0,000	0,000	0,000	0,027	0,000	0,026	0,000

Table 5.16: Function-level metrics Weight.

	(a) Linux Kernel				(b) Mozilla Firefox			
Software Metrics	**RF**	**DT**	**Xboost**	**SVM**	**RF**	**DT**	**Xboost**	**SVM**
CountOutput	0,071	0,044	0,076	0,056	0,055	0,000	0,089	0,053
CountLineCodeDecl	0,062	0,000	0,054	0,048	0,044	0,000	0,063	0,052
MaxNesting	0,043	0,024	0,000	0,000	0,050	0,000	0,034	0,052
CountInput	0,042	0,039	0,068	0,050	0,042	0,000	0,077	0,000
AltCountLineBlank	0,040	0,004	0,050	0,049	0,045	0,207	0,065	0,055
Knots	0,040	0,000	0,047	0,051	0,025	0,000	0,026	0,040
CountLineBlank	0,039	0,005	0,000	0,049	0,052	0,209	0,086	0,055
CountLineCode	0,036	0,163	0,133	0,056	0,039	0,196	0,056	0,053
MinEssentialKnots	0,036	0,182	0,041	0,049	0,024	0,000	0,000	0,000
AltCountLineComment	0,036	0,005	0,031	0,000	0,037	0,000	0,029	0,045
MaxEssentialKnots	0,036	0,180	0,038	0,049	0,025	0,000	0,000	0,000
CyclomaticStrict	0,035	0,007	0,027	0,044	0,026	0,000	0,025	0,050
CountSemicolon	0,034	0,000	0,044	0,047	0,031	0,000	0,038	0,047
CountLineComment	0,034	0,000	0,000	0,000	0,037	0,000	0,000	0,045
CountStmtDecl	0,034	0,000	0,035	0,000	0,042	0,000	0,032	0,048
Cyclomatic	0,034	0,000	0,000	0,043	0,033	0,000	0,000	0,051
CountLine	0,033	0,156	0,059	0,056	0,042	0,196	0,103	0,055
CountLineCodeExe	0,033	0,000	0,045	0,056	0,035	0,000	0,050	0,051
CyclomaticModified	0,033	0,000	0,025	0,044	0,034	0,191	0,020	0,052
RatioCommentToCode	0,033	0,005	0,033	0,000	0,045	0,000	0,042	0,000
CountPath	0,033	0,000	0,049	0,050	0,024	0,000	0,045	0,049
AltCountLineCode	0,033	0,160	0,065	0,056	0,040	0,000	0,045	0,052
CountStmtExe	0,032	0,000	0,039	0,048	0,040	0,000	0,040	0,047
CountStmt	0,031	0,000	0,038	0,048	0,039	0,000	0,033	0,048
Essential	0,031	0,026	0,000	0,049	0,028	0,000	0,000	0,000
CountLinePreprocessor	0,021	0,000	0,000	0,000	0,007	0,000	0,000	0,000
CountLineInactive	0,019	0,000	0,000	0,000	0,006	0,000	0,000	0,000
CountStmtEmpty	0,018	0,000	0,000	0,000	0,052	0,000	0,000	0,000

Table 5.17: Example of the real (a) and normalized (b) values of four software metrics for five Linux Kernel's files.

(a) File-level dataset of Linux Kernel project

No.	**ID File**	**Label**	**Count Line Inactive**	**Sum Cyclomatic**	**Fan Out**	**HK**
1	**1407302**	v	1	4	12	150
2	**6081702**	v	0	46	164	113172372
3	**15866802**	nv	141	113	22	13821104
4	**2627502**	v	7	203	270	23115237
5	**32375002**	nv	302	385	148	1933014887

(b) Normalized File-level dataset of Linux Kernel project

No.	**ID File**	**Label**	**Count Line Inactive**	**Sum Cyclomatic**	**Fan Out**	**HK**
1	**1407302**	v	0,993	0,991	0,900	1,000
2	**6081702**	v	1,000	0,898	0,000	0,000
3	**15866802**	nv	0,000	0,750	0,817	0,126
4	**2627502**	v	0,948	0,550	0,000	0,000
5	**32375002**	nv	0,000	0,147	0,000	0,000

For each cluster (in Table 5.19), the number of files and function included and the *Medoids* value of their score are also presented. The *Medoids* represents the relative final score of each cluster; thus, it can be used to rank and label the clusters from trustworthy to absolutely untrustworthy (i.e., Cluster 1 is considered as trustworthy and Cluster 5 is considered as absolutely untrustworthy).

Interestingly, in all cases, the number of files and functions included in each cluster decreases when the value of *Medoids* decreases. It means that less trust-

Table 5.18: Example of trustworthiness scores for five files of Linux Kernel.

		Trustworthiness Score			
#	ID File	RF	DT	Xboost	SVM
1	**1407302**	0,904	0,975	0,903	0,960
2	**6081702**	0,548	0,800	0,812	0,811
3	**15866802**	0,570	0,600	0,585	0,650
4	**2627502**	0,500	0,518	0,538	0,533
5	**32375002**	0,295	0,134	0,161	0,242

worthy clusters have fewer code units. For instance, 27156 files (28%) belong to the cluster with a higher level of trustworthiness, and 12380 files (13%) belong to the cluster with the lowest trustworthiness level (combination of all scores in Table 5.19 (a)).

Considering the clustering results for Mozilla Firefox files and functions (presented in Table 5.20), the main observations are similar to the ones discussed for Linux Kernel. In fact the number of files and functions decreases as we move from cluster 1 to cluster 5. Again, the results suggest that less trustworthy clusters have fewer code units. We again observed that there is no major difference between the results of clustering on individual scores (considering one single trustworthiness model) and the results of clustering on the combined scores (considering several trustworthiness models by aggregating all scores).

5.4.2.2 Assessment of Linux Kernel results

To study the applicability of the proposed approach and the accuracy of the results, we again used the security vulnerabilities of the Linux Kernel project reported after 2017 (recall that the dataset used in this experiment only contains data from 2000 to 2016). That data was not added to the dataset for training the models, as we wanted to use them for validation. As mentioned before, the vulnerabilities were collected from CVEDetails and we identified a total of 801 files and 650 functions with vulnerabilities reported after 2017.

The validation consisted of verifying in which cluster each of these files/functions was classified. In other words, for each cluster, we counted the number of unique files/functions with reported vulnerabilities after 2017. The assumption is that, if a specific cluster has more reported vulnerabilities than the others, then the cluster (potentially) includes code units more prone to be vulnerable.

Table 5.21 presents the results of the five clustering experiments, of which four are on individual scores and one is on the combination of all scores. Each table presents the total number of files and the total number of vulnerable files, and their percentages in each cluster. As we can see:

- The percentage of vulnerable files increases as we move from cluster 1 to cluster 5. For instance, observing the results using the score of RF (Table 5.21 (a)), the less trustworthy cluster (Cluster 5, having a *Medoid* value of 0.304, as shown in Table 5.19 (a)) has the highest percentage of files discovered as vulnerable in 2017 or after (9.9%);

Table 5.19: Clustering results of Linux Kernel project.

(a) File-level results

	Score RF		**Score DT**		**Score Xboost**		**Score SVM**		**Score RF, Score DT, Score Xboost, Score SVM**			
Clusters	**# Files**	**Medoid**	**# Files**	**Medoid**	**# Files**	**Medoid**	**# Files**	**Medoid**	**# Files**	**Med. RF**	**Med. DT**	**Med. Xboost**	**Med. SVM**
Cluster 1	22813	0,85	28581	0,933	24769	0,883	26508	0,928	27156	0,830	0,927	0,884	0,927
Cluster 2	22793	0,758	25838	0,801	22591	0,795	25721	0,801	25098	0,738	0,815	0,759	0,802
Cluster 3	21940	0,637	18195	0,631	20851	0,67	18675	0,65	17429	0,578	0,657	0,644	0,65
Cluster 4	15651	0,478	12645	0,404	15400	0,464	13369	0,435	13842	0,478	0,449	0,475	0,478
Cluster 5	12708	0,304	10646	0,076	12294	0,182	11632	0,136	12380	0,337	0,117	0,155	0,148
Total	95905	-	95905	-	95905	-	95905	-	95905	-	-	-	-

(b) Function-level results

	Score RF		**Score DT**		**Score Xboost**		**Score SVM**		**Score RF, Score DT, Score Xboost, Score SVM**			
Clusters	**# Func.**	**Medoid**	**# Func.**	**Medoid**	**# Func.**	**Medoid**	**# Func.**	**Medoid**	**# Func.**	**Med. RF**	**Med. DT**	**Med. Xboost**	**Med. SVM**
Cluster 1	135767	0,885	189464	0,928	152339	0,930	173559	0,921	166562	0,876	0,935	0,923	0,924
Cluster 2	115832	0,782	101008	0,817	124464	0,798	113869	0,788	132604	0,756	0,809	0,778	0,797
Cluster 3	95082	0,639	79247	0,650	88195	0,642	78533	0,631	83694	0,571	0,566	0,570	0,583
Cluster 4	73553	0,448	62324	0,351	65060	0,405	62298	0,405	51796	0,386	0,286	0,326	0,343
Cluster 5	57108	0,161	45299	0,045	47284	0,088	49083	0,086	42686	0,133	0,043	0,120	0,084
Total	477342	-	477342	-	477342	-	477342	-	477342	-	-	-	-

Table 5.20: Clustering results of Mozilla Firefox project.

(a) File-level results

	Score RF		Score DT		Score Xboost		Score SVM		Score RF, Score DT, Score Xboost, Score SVM				
Clusters	**# Files**	**Medoid**	**# Files**	**Medoid**	**# Files**	**Medoid**	**# Files**	**Medoid**	**# Files**	**Med. RF**	**Med. DT**	**Med. Xboost**	**Med. SVM**
Cluster 1	14094	0,834	19037	0,932	16220	0,886	16185	0,949	15580	0,834	0,939	0,893	0,948
Cluster 2	10991	0,724	10314	0,804	11638	0,778	12069	0,833	11595	0,716	0,848	0,795	0,827
Cluster 3	9068	0,622	6661	0,635	7386	0,640	7270	0,646	8344	0,615	0,689	0,640	0,686
Cluster 4	7404	0,455	4928	0,367	5568	0,445	5014	0,417	5336	0,455	0,342	0,447	0,380
Cluster 5	4887	0,256	5504	0,000	5632	0,148	5906	0,038	5589	0,307	0,000	0,147	0,043
Total	46444	-	46444	-	46444	-	46444	-	46444	-	-	-	-

(b) Function-level results

	Score RF		Score DT		Score Xboost		Score SVM		Score RF, Score DT, Score Xboost, Score SVM				
Clusters	**# Func.**	**Medoid**	**# Func.**	**Medoid**	**# Func.**	**Medoid**	**# Func.**	**Medoid**	**# Func.**	**Med. RF**	**Med. DT**	**Med. Xboost**	**Med. SVM**
Cluster 1	122468	0,833	134553	0,954	139925	0,884	131761	0,940	122019	0,844	0,954	0,900	0,949
Cluster 2	86253	0,748	78150	0,841	80185	0,755	83875	0,819	80506	0,750	0,853	0,788	0,829
Cluster 3	59840	0,580	60635	0,663	54522	0,609	56223	0,645	54447	0,630	0,709	0,651	0,682
Cluster 4	42818	0,389	40298	0,369	37772	0,374	40862	0,403	48038	0,424	0,507	0,462	0,516
Cluster 5	43101	0,092	40844	0,000	42076	0,101	41759	0,067	49470	0,112	0,036	0,133	0,114
Total	354480	-	354480	-	354480	-	354480	-	354480	-	-	-	-

Table 5.21: Validation of clustering results using files of Linux project.

(a) Score RF

Clusters	**# Files**	**% Files**	**# vuln.**	**% vuln.**
Cluster 1	22813	23,8%	79	0,3%
Cluster 2	22793	23,8%	282	1,2%
Cluster 3	21940	22,9%	643	2,9%
Cluster 4	15651	16,3%	855	5,5%
Cluster 5	12708	13,3%	1264	9,9%
Total	95905	-	3123	-

(b) Score DT

Clusters	**# Files**	**% Files**	**# vuln.**	**% vuln.**
Cluster 1	28581	29,8%	102	0,4%
Cluster 2	25838	26,9%	466	1,8%
Cluster 3	18195	19,0%	519	2,9%
Cluster 4	12645	13,2%	850	6,7%
Cluster 5	10646	11,1%	1186	11,1%
Total	95905	-	3123	-

(c) Score Xboost

Clusters	**# Files**	**% Files**	**# vuln.**	**% vuln.**
Cluster 1	24769	25,8%	82	0,3%
Cluster 2	22591	23,6%	279	1,2%
Cluster 3	20851	21,7%	603	2,9%
Cluster 4	15400	16,1%	816	5,3%
Cluster 5	12294	12,8%	1343	10,9%
Total	95905	-	3123	-

(d) Score SVM

Clusters	**# Files**	**% Files**	**# vuln.**	**% vuln.**
Cluster 1	26508	27,6%	66	0,2%
Cluster 2	25721	26,8%	424	1,6%
Cluster 3	18675	19,5%	592	3,2%
Cluster 4	13369	13,9%	820	6,1%
Cluster 5	11632	12,1%	1221	10,5%
Total	95905	-	3123	-

(e) Score RF, Score DT, Score Xboost, Score SVM

Clusters	**# Files**	**% Files**	**# vuln.**	**% vuln.**
Cluster 1	27156	28,3%	79	0,29%
Cluster 2	25098	26,2%	389	1,55%
Cluster 3	17429	18,2%	555	3,18%
Cluster 4	13842	14,4%	775	5,60%
Cluster 5	12380	12,9%	1325	10,70%
Total	95905	-	3123	-

Table 5.22: Validation of clustering results using functions of Linux project.

(a) Score RF

Clusters	**# Funcs.**	**% Funcs.**	**# vuln.**	**% vuln.**
Cluster 1	135767	28,4%	37	0,03%
Cluster 2	115832	24,3%	129	0,11%
Cluster 3	95082	19,9%	108	0,11%
Cluster 4	73553	15,4%	153	0,21%
Cluster 5	57108	12,0%	175	0,31%
Total	477342	-	602	-

(b) Score DT

Clusters	**# Funcs.**	**% Funcs.**	**# vuln.**	**% vuln.**
Cluster 1	189464	39,7%	65	0,03%
Cluster 2	79247	16,6%	108	0,14%
Cluster 3	101008	21,2%	160	0,16%
Cluster 4	62324	13,1%	108	0,17%
Cluster 5	45299	9,5%	161	0,36%
Total	477342	-	602	-

(c) Score Xboost

Clusters	**# Funcs.**	**% Funcs.**	**# vuln.**	**% vuln.**
Cluster 1	152339	31,9%	45	0,03%
Cluster 2	124464	26,1%	152	0,12%
Cluster 3	88195	18,5%	113	0,13%
Cluster 4	65060	13,6%	132	0,20%
Cluster 5	47284	9,9%	160	0,34%
Total	477342	-	602	-

(d) Score SVM

Clusters	**# Funcs.**	**% Funcs.**	**# vuln.**	**% vuln.**
Cluster 1	173559	36,4%	61	0,04%
Cluster 2	78533	16,5%	89	0,11%
Cluster 3	113869	23,9%	156	0,14%
Cluster 4	62298	13,1%	131	0,21%
Cluster 5	49083	10,3%	165	0,34%
Total	477342	-	602	-

(e) Score RF, Score DT, Score Xboost, Score SVM

Clusters	**# Funcs.**	**% Funcs.**	**# vuln.**	**% vuln.**
Cluster 1	166562	34,9%	57	0,03%
Cluster 2	132604	27,8%	170	0,13%
Cluster 3	83694	17,5%	127	0,15%
Cluster 4	51796	10,9%	94	0,18%
Cluster 5	42686	8,9%	154	0,36%
Total	477342	-	602	-

- In contrast, the cluster with the highest level of trustworthiness (cluster 1, with a *Medoid* value of 0.85) has the lowest percentage of recently vulnerable files (0.3%);
- Interestingly, in all cases, the percentage of recent vulnerable files increases when the cluster becomes less trustworthy. The same observation is true in the case of Linux kernel functions presented in Table 5.22;
- The results show that, in fact, the files and functions categorized as absolutely untrustworthy are more prone to be problematic.

A key observation of our study suggested that there is no major difference between the results of clustering on individual scores (considering one single trustworthiness model) and the results of clustering on the combined scores (considering several trustworthiness models by aggregating all scores). This may question the principal idea of our proposed approach by creating doubts on the need to consider various scores when the same clustering results can be achieved using only one score. This is true in the present study as we used the same scoring model in all five models, i.e., the same dataset for creating machine learning-based models built on the same evidence of software quality, i.e., the software metrics. Nevertheless, our overall approach is neither limited to machine learning nor to software metrics. Any evidence of security issues and any scoring model can be used to assign different scores to each unit of code.

5.4.2.3 Assessment of Mozilla Firefox results

To validate the results obtained for the Mozilla Firefox project, we again resort to the expert-based ranking performed in Section 4.4, were we asked the experts to rank 10 files and 10 functions based on their perceived trustworthiness.

The information for the files and functions used for validation is in Table 5.23, in which we present the ranking and scores given by experts and compare them with the average *Medoids* of the cluster to which each file belongs.

Table 5.23: Validation of clustering Results using Mozilla files and functions.

ID File	Experts based results	Average Medoids	ID Function	Experts based results	Average Medoids
60426601	0,429	0,912	361442101	0,462	0,919
263601	0,408	0,912	20082901	0,458	0,919
1701201	0,31	0,912	304227601	0,272	0,919
51355601	0,233	0,912	121633001	0,303	0,810
52292101	0,165	0,803	311237201	0,153	0,671
1418301	0,171	0,663	113275001	0,082	0,671
2201801	0,103	0,663	119075401	0,094	0,485
45301201	0,07	0,108	314376601	0,057	0,485
22256501	0,065	0,108	328555501	0,055	0,485
19960501	0,046	0,108	53225001	0,063	0,102

The table is ordered from the highest to the lowest Average Medoid value and we can clearly observe that the order of files and functions is similar. In fact, the files are ordered equally by experts based ranking and using the average medoids (clustering process). For functions, the results are very similar, except for two

functions (ID Function number 121633001 and 119075401). By observing the source code of those two functions, we noticed that they are very small and simple (with 18 and 38 lines of code, respectively). We believe that this led to a higher assessment of trustworthiness in the expert based results.

The results obtained clearly suggest that **our approach is able to provide a meaningful categorization of the files and functions**.

5.4.3 Considering Different Number of Clusters

The number of files and functions included in the less trustworthy clusters can be high in some application scenarios. Thus, it can be helpful to have a higher number of clusters, allowing the developers to more conveniently chose the groups to be worked on. For this purpose, we performed an experiment to observe the impact of the number of clusters on the results.

We used the trustworthiness scores obtained using all trustworthiness models for clustering files and functions into 3 and 7 clusters, to be compared with the previously obtained results with 5 clusters. Table 5.24 presents the results for Linux kernel files (a) and Linux Kernel functions (b). The results obtained (with 3 and 7 clusters) are on par with the previous ones (with 5 clusters): the less trustworthy a category (or cluster) is, the higher proportion of the code units included have known vulnerabilities.

We observed that it is possible to categorize the code in a more precise manner by increasing the number of clusters, allowing to achieve a higher intra-cluster similarity and lower inter-cluster similarity. This means that increasing the number of clusters resulted in smaller clusters (with fewer code units) but still with an appropriate (with respect to the total number of code units in each cluster and considering their their trustworthiness level) number of code units with known vulnerabilities (e.g., the least trustworthy category includes a higher percentage of vulnerable code, i.e., 11.6 % for files data). Therefore, when there are fewer resources to review code and eliminate vulnerabilities, it is quite useful to perform the clustering with a higher number of clusters, which decreases the number of code units that must be reviewed and, at the same time, allows dealing with a reasonable number of vulnerabilities.

To better understand the characteristics of the code units in each cluster, we calculated the average value of two key software file and function-level metrics: *CountLineCode* and *Cyclomatic* complexity for functions and *CountLineCode* and *SumCyclomatic* for files. Interestingly, in the case of files, the value of both *CountLineCode* and *SumCyclomatic* metrics increases when the level of trustworthiness decreases. This continues to be true even if we increase the number of clusters. In the case of functions, similar results are observed with 5 clusters. However, when the code units are categorized into 3 or 7 clusters, we cannot see big differences between the average values of the two metrics, which implies that other metrics rather than the volume and complexity metrics (e.g., *MaxNesting*) were influential in the trustworthiness level of functions. However, in most cases, the results show that when the size and complexity increase, the

Table 5.24: Results using different numbers of clusters.

(a) File-level

	Number of Clusters = 3						Number of Clusters = 5						Number of Clusters = 7					
Clusters	**# Files**	**% Files**	**# Vuln**	**% Vuln**	**CountLineCode**	**SumCyclomatic**	**# Files**	**% Files**	**# Vuln**	**% Vuln**	**CountLineCode**	**SumCyclomatic**	**# Files**	**% Files**	**# Vuln**	**% Vuln**	**CountLineCode**	**SumCyclomatic**
Cluster 1	43,697	45.6%	288	0.7%	183.3	30.7	27,156	28.3%	79	0.3%	119.7	17.4	19,190	20%	46	0.2%	93.4	12.0
Cluster 2	33,324	34.7%	1060	3.2%	639.2	147	25,098	26.2%	389	1.6%	332.8	62.6	18,751	19.6%	168	0.9%	226.5	39.4
Cluster 3	18,884	19.7%	1,775	9.4%	2050.5	613.2	17,429	18.2%	555	3.2%	623.1	141.4	19,002	19.8%	397	2.09%	413.2	81.7
Total	95,905	-	3,123	-	-	-	13,842	14.4%	775	5.6%	1073.7	277.6	13,906	14.5%	459	3.3%	687.5	158.6
							1,238	12.9%	1,325	10.7%	2479.3	764.9	9,195	9.6%	459	4.99%	1010.4	259.0
							Total	95905	3,123	-	-	-	8,367	8.7%	722	8.63%	1520.4	463.5
													7,494	7.8%	872	11.6%	3010	911.8
													Total	95,905	3,123	-	-	-

(b) Function-level

	Number of Clusters = 3						Number of Clusters = 5						Number of Clusters = 7					
Clusters	**# Files**	**% Files**	**# Vuln**	**% Vuln**	**CountLineCode**	**Cyclomatic**	**# Files**	**% Files**	**# Vuln**	**% Vuln**	**CountLineCode**	**Cyclomatic**	**# Files**	**% Files**	**# Vuln**	**% Vuln**	**CountLineCode**	**Cyclomatic**
Cluster 1	43,697	45.6%	79	0.04%	32.0	9.2	27,156	28.3%	57	0.03%	6.9	1.7	19,190	20%	30	0.03%	33.6	9.5
Cluster 2	33,324	34.7%	75	0.05%	24.8	9.4	25,098	26.2%	170	0.13%	17.8	5.6	18,751	19.6%	22	0.04%	19.8	6.9
Cluster 3	18,884	19.7%	448	0.26%	30.9	9.6	17,429	18.2%	127	0.15%	32.8	10.7	19,002	19.8%	37	0.05%	31.9	9.7
Total	95,905	-	602	-	-	-	13,842	14.4%	94	0.18%	55.9	19.3	13,906	14.5%	30	0.05%	29.5	11.6
							1,238	12.9%	154	0.36%	116.2	37.0	9,195	9.6%	35	0.05%	25.9	8.9
							Total	95905	602	-	-	-	8,367	8.7%	166	0.21%	33.7	9.8
													7,494	7.8%	282	0.36%	27.6	9.0
													Total	95,905	602	-	-	-

trustworthiness level decreases.

An interesting observation is that, when we compare the characteristics of the vulnerable/non-vulnerable files wrongly classified by all models with the characteristics of the ones that are correctly classified by all models (refer to Figure 3.23 in Section 3.3.5.2), we conclude that correctly classified vulnerable files (TPs) and wrongly classified non-vulnerable files (FPs) are typically bigger and more complex. On the other hand, correctly classified non-vulnerable files (TNs) and wrongly classified vulnerable files (FNs) are normally smaller and simpler considering their structural characteristics.

5.5 Summary

In this chapter, we proposed a **framework for code categorization based on security evidences**. The idea is to define a mechanism that, based on the early evidences of security issues in the code, supports developers in the detection of potential issues during the software development process. Two instantiations of the framework were presented, both of them based on **machine learning algorithms** (prediction models) and using **software metrics** as security evidences: *i)* a Consensus-Based Decision-Making (CDBM) approach ; and *ii)* an approach based on Trustworthiness Models (TMs).

By applying the first approach, we were able to prioritize categories from a security perspective, considering different scenarios. The results show that the consensus-based decision-making solution is more useful and suitable for diverse application scenarios than a single prediction model. Considering the second approach, the results show that we can effectively use clustering in order to identify code units more prone to be vulnerable. Thus, developers can use this approach to make efficient and effective decisions about the parts of code that might be problematic and require deep analysis. Besides that, it is possible to adjust the number of clusters considering different scenarios, where different resources to detect and eliminate vulnerabilities are available.

We highlight the main insights obtained from the results:

- The number of files and functions included in each cluster decreases when the trustworthiness level decreases. It means that less trustworthy clusters have fewer code units;
- The percentage of new reported vulnerabilities increases as we move from trustworthy cluster to absolutely untrustworthy cluster. This strongly support the validation of the proposed clustering-based approach;
- By increasing the clusters, a higher intra-cluster similarity is achieved. This means that increasing the number of clusters resulted in smaller clusters (with fewer code units) with more similar characteristics in terms of software metrics and trustworthiness level. Thus, when there are fewer resources to review code and eliminate vulnerabilities, it is quite useful to perform the clustering with a higher number of clusters.

Although the results show that we can effectively use our approach to make effective decisions about the parts of code that might be problematic, some limitations and threats to validity should be highlighted:

- The prediction models are built using Machine Learning algorithms and software metrics are used as evidence of security issues in code. Although our experimental evaluations show the applicability of the proposed framework, future work should focus on using different tools and techniques as scoring models, as well as other security evidences (e.g., code smells);
- The number of files and functions selected to validate the results for the Mozilla Firefox project is limited (10 files and 10 functions). This is a feasible acceptable number of files and functions, since the experts had to perform the comparison between all the possible pairs. However, it would be better to get a larger number of compared code units;
- The dataset used for instantiating the proposed approach is limited to two projects. Although the projects are quite big (with a large number of files and functions used to build the trustworthiness models) and representative for this study, including more projects could help achieving more convincing results.

According to the main insights from the results, we can conclude that the idea of **grouping code units into several categories representing their trustworthiness level from a security perspective** can be used in order to call the attention of developers to the most untrustworthy code units. In fact, we consider that the proposed framework for code units categorization and its instantiations are indeed suitable to support developers in the identification of the code units more prone to be vulnerable from the early phases of coding process.

Chapter 6

Conclusions and Future Work

This thesis discussed the use of evidence collected from software to identify potentially vulnerable code units. The main objective is to advance the state-of-the-art on tools and techniques for improving software security, in a way that help development teams from the early phases of software development process.

Two comprehensive case studies on how software metrics and Machine Learning can be used to predict/detect vulnerable code units were presented. Then, a trustworthiness benchmarking framework based on software metrics were proposed, focusing on prioritizing code units based on their perceived trustworthiness. Finally, two alternative methodologies to identify code units more prone to be vulnerable were proposed. The main goal was to define mechanisms that, based on evidence of security practices and issues in the code, are able to categorize code into different trustworthiness levels.

6.1 Conclusions

The thesis began with an exploratory and empirical analysis focusing on the possibility of finding the best subset of software metrics for building the most accurate classifier model to distinguish vulnerable from non-vulnerable code units. To do so, a set of different experiments were conducted, namely: *(i) a statistical correlation analysis* using project-level metrics and security vulnerabilities; *(ii) a dimension reduction* that contributes to select different groups of software metrics; and *(iii) a feature selection analysis* using the Genetic Algorithm and the Random Forest classifier.

The results showed that there is a strong correlation between several project-level metrics and the number of reported vulnerabilities, and it is possible to use a group of metrics to distinguish vulnerable and non-vulnerable units of code with a high level of accuracy. However, the best subset of predictive metrics

may vary from one software system to another. Besides that, for identifying the quality of software in terms of security, the function, file and project level metrics are complementary to each other.

The next contribution of the thesis was a comprehensive experiment to study how effective software metrics and Machine Learning algorithms are for classifying vulnerable and non-vulnerable units of code. Machine Learning models were trained considering metrics at different architectural levels and several software projects. The most important observation is that using Machine Learning algorithms on top of software metrics helps identifying vulnerable code units with relatively high level of confidence for security-critical software systems (where the focus is on detecting the maximum number of vulnerabilities, even if false positives are reported), but they are not helpful for low-critical or non-critical systems due to the relatively high number of false positive alarms reported when compared to the number of true positive cases (that bring an additional development cost frequently not affordable), which is mainly caused by the imbalanced nature of our dataset (and similar datasets used in other works).

According to our observations, insights and threats to the validity of the two case studies, we can conclude that software metrics are not sufficient evidence of security issues to be used solely for building detection/prediction models that are able to distinguish vulnerable code from non-vulnerable code with good performance and low vulnerability removal cost. Moreover, due to the natural limitations of existing datasets for training and testing these models, it becomes even more difficult to precisely understand how effective software metrics can be to detect vulnerable code in different application scenarios.

Based on this strong conclusion, we next focused on using software metrics not for predicting or detecting vulnerabilities but for assessing/benchmarking the trustworthiness of the code and warn the developers about their untrustworthy (insecure) code units. Therefore, we move from a subjective to an objective trustworthiness notion and proposed a trustworthiness model directly by using a group of software metrics that were weighted based on the scores given by a classification model. The output of the proposed benchmark are the trustworthiness scores assigned to each code unit. Results show that the proposed benchmark enables the characterization of units of code and enables their comparison. In fact, the trustworthiness benchmarking results were assessed by comparing the trustworthiness scores assigned to different files and functions (i.e., ranking of files and functions considering the trustworthiness scores given by the benchmarking approach) with an expert-based ranking. We observed a sound ranking of the benchmarked files and functions. Note that, although our main goal was to build a trustworthiness assessment model based on security evidences, the framework can be used to benchmark any other attributes of software quality (e.g., privacy and performance).

Building on top of the results of the proposed benchmarking approach, we next proposed an approach to identify code units that may be more prone to be vulnerable, again based on software metrics (security evidence) and machine learning algorithms. Two different instantiations were presented, the first groun-

ded on a Consensus-Based Decision-Making approach, and the second based on Trustworthiness Models to categorize code.

Results show that the two approaches are able to prioritize and categorize code units from a security perspective, considering different scenarios. In fact, we showed that both solutions are more useful and suitable for diverse application scenarios than a single prediction model, and that the percentage of new reported vulnerabilities increases as we move from trustworthy to absolutely untrustworthy units of code. Based on this, we strongly believe that developers can use our approaches to make more efficient and effective decisions about the parts of code that might be problematic and require a deeper analysis.

It is important to emphasize that, the proposed framework can be instantiated using different approaches. For example, any other kind of security evidences (e.g., code smells) or other quality attributes (instead of security) can be considered. Besides that, other prediction models or techniques to predict/detect vulnerabilities can be applied as characterization models, and different categorization mechanism (besides grouping classification results of ML and perform clustering) can be implemented.

6.2 Future work

Building on the results obtained, the important conclusions and contributions, this thesis opens the door for several future works:

- Study the applicability of other ML algorithms and different artificial intelligence techniques (such as neural networks) to predict/detect vulnerabilities, to instantiate the proposed framework for code units categorization.
- Use other evidences rather than software metrics, like code smells , lack of security best practices in the code, alerts given by static code analysers, among others, to improve the detection/prediction models to produce less false alarms. Also, consider different quality attributes besides than security.
- Explore the use of semi-supervised techniques to find natural groupings of the data for building the classifiers. Semi-supervised approaches should be studied as alternative choices where is not trivial to verify the label of all records.
- Different projects with different characteristics in terms of structure and implemented in different languages should be used.
- Apply the proposed code units categorization framework in a real development team working context of continuous integration. However, managing and performing such experiments is quite challenging.

Bibliography

Abdlhamed, M., Kifayat, K., Shi, Q., and Hurst, W. (2017). Intrusion prediction systems. In *Information Fusion for Cyber-Security Analytics*, pages 155–174. Springer.

Acar, Y., Stransky, C., Wermke, D., Weir, C., Mazurek, M. L., and Fahl, S. (2017). Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26. IEEE.

Agrawal, A. and Khan, R. (2009). A framework to detect and analyze software vulnerabilities: Development phase perspective. *International Journal of Recent Trends in Engineering*, 2.

Alarcon, G. M., Gamble, R., Jessup, S. A., Walter, C., Ryan, T. J., Wood, D. W., and Calhoun, C. S. (2017a). Application of the heuristic-systematic model to computer code trustworthiness: The influence of reputation and transparency. *Cogent Psychology*, 4(1):1389640.

Alarcon, G. M., Militello, L. G., Ryan, P., Jessup, S. A., Calhoun, C. S., and Lyons, J. B. (2017b). A descriptive model of computer code trustworthiness. *Journal of Cognitive Engineering and Decision Making*, 11(2):107–121.

Alenezi, M. and Zarour, M. (2018). Software vulnerabilities detection based on security metrics at the design and code levels: empirical findings. *Journal of Engineering Technology*, 6(1):570–583.

Alhazmi, O. H., Malaiya, Y. K., and Ray, I. (2007). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219–228.

Alves, H., Fonseca, B., and Antunes, N. (2016a). Experimenting machine learning techniques to predict vulnerabilities. In *Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156.

Alves, H., Fonseca, B., and Antunes, N. (2016b). Software metrics and security vulnerabilities: Dataset and exploratory study. In *12th European Dependable Computing Conference (EDCC)*, pages 37–44. IEEE.

Amoroso, E. G., Taylor, C. A., Watson, J., and Weiss, J. (1994). A process-oriented methodology for assessing and improving software trustworthiness. In *CCS '94*.

Ankrum, T. and Kromholz, A. (2005). Structured assurance cases: three common standards. In *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*, pages 99–108.

Antunes, N. and Vieira, M. (2009). 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.

Antunes, N. and Vieira, M. (2010). Benchmarking vulnerability detection tools for web services. In *IEEE International Conference on Web Services (ICWS)*, pages 203–210. IEEE.

Antunes, N. and Vieira, M. (2015). On the metrics for benchmarking vulnerability detection tools. In *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 505–516. IEEE.

Arkin, B., Stender, S., and McGraw, G. (2005). Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87.

Artz, D. and Gil, Y. (2007). A survey of trust in computer science and the semantic web. *Journal of Web Semantics*, 5(2):58–71. Software Engineering and the Semantic Web.

Aruldoss, M., Lakshmi, T. M., and Venkatesan, V. P. (2013). A survey on multi criteria decision making methods and its applications. *American Journal of Information Systems*, 1(1).

Assal, H. and Chiasson, S. (2018). Security in the software development lifecycle.

Assal, H. and Chiasson, S. (2019). *'Think Secure from the Beginning': A Survey with Software Developers*, page 1–13. Association for Computing Machinery, New York, NY, USA.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

Awad, M. and Khanna, R. (2015). Support vector machines for classification - efficient learning machines: Theories, concepts, and applications for engineers and system designers. In *Efficient Learning Machines*, pages 39–66. Apress.

Bardas, A. G. (2010). Static Code Analysis. *Romanian Economic Business Review*, 4(2):99–107.

Barioni, M., Razente, H., Marcelino, A., Traina, A., and Jr, C. (2014). Open issues for partitioning clustering methods: An overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4.

Bashir, U. and Chachoo, M. (2014). Intrusion detection and prevention system: Challenges opportunities. In *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 806–809.

Batista, G. E., Prati, R. C., and Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD explorations newsletter*, 6(1):20–29.

Beckers, K., Côté, I., Fenz, S., Hatebur, D., and Heisel, M. (2014). A structured comparison of security standards. In *Engineering secure future internet services and systems*, pages 1–34. Springer.

Benesty, J., Chen, J., Huang, Y., and Cohen, I. (2009). Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer.

Bommert, A., Sun, X., Bischl, B., Rahnenführer, J., and Lang, M. (2020). Benchmark for filter methods for feature selection in high-dimensional classification data. *Computational Statistics & Data Analysis*, 143:106839.

Borkin, D., Némethová, A., Michal'čonok, G., Maiorov, K., et al. (2019). Impact of data normalization on classification model accuracy. *Research Papers Faculty of Materials Science and Technology Slovak University of Technology*, 27(45):79–84.

Boser, B., Guyon, I., and Vapnik, V. (1992). A training algorithm for optimal margin classifiers. In *COLT '92 Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM.

Brar, H. and Kaur, P. (2015). Comparing detection ratio of three static analysis tools. *International Journal of Computer Applications*, 124.

Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., et al. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Briand, L. C. and et. al. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3):245–273.

Cairo, A. S., Carneiro, G. d. F., and Monteiro, M. P. (2018). The impact of code smells on software bugs: A systematic literature review. *Information*, 9(11):273.

Calle, M. L. and Urrea, V. (2010). Letter to the editor: stability of random forest importance measures. *Briefings in bioinformatics*, 12(1):86–89.

Campbell, G. and Papapetrou, P. P. (2013). *SonarQube in action*. Manning Publications Co.

Cao, L. (2012). Dynamic capability for trustworthy software development. *Journal of Software: Evolution and Process*, 24(7):837–850.

Cavoukian, A. (2009). Privacy by design - the 7 foundational principles - implementation and mapping of fair information practices. *Information & Privacy Commissioner, Ontario, Canada*.

Chawla, N. V., Japkowicz, N., and Kotcz, A. (2004). Special issue on learning from imbalanced data sets. *ACM Sigkdd Explorations Newsletter*, 6(1):1–6.

Chemuturi, M. (2010). *Mastering software quality assurance: best practices, tools and techniques for software developers*. J. Ross Publishing.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794.

Chess, B. and McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79.

Chillarege, R. and et. al. (1992). Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering*, 18(11):943–956.

Chillarege, R. et al. (1996). Orthogonal defect classification. *Handbook of software reliability engineering*, pages 359–399.

Chiregi, M. and Navimipour, N. (2017). A comprehensive study of the trust evaluation mechanisms in the cloud computing. *Journal of Service Science Research*, 9:1–30.

Cho, J.-H., Chan, K., and Adali, S. (2015). A survey on trust modeling. *ACM Comput. Surv.*, 48(2).

Cho, J.-H., Hurley, P. M., and Xu, S. (2016). Metrics and measurement of trustworthy systems. In *MILCOM 2016 - 2016 IEEE Military Communications Conference*, pages 1237–1242.

Chowdhury, I. and Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

Crawford, G. and Williams, C. (1985). A note on the analysis of subjective judgment matrices. *Journal of mathematical psychology*, 29(4):387–405.

Criteria, C. (1999). Common criteria for information technology security evaluation.

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

De Cremer, P., Desmet, N., Madou, M., and De Sutter, B. (2020). Sensei: Enforcing secure coding guidelines in the integrated development environment. *Software: Practice and Experience*, 50(9):1682–1718.

De Landgraaf, W., Eiben, A., and Nannen, V. (2007). Parameter calibration using meta-algorithms. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 71–78. IEEE.

Del Bianco, V., Lavazza, L., Morasca, S., and Taibi, D. (2011). A survey on open source software trustworthiness. *IEEE software*, 28(5):67–75.

Deswarte, Y., Blain, L., and Fabre, J.-C. (1991). Intrusion tolerance in distributed computing systems. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy*, pages 110–121. IEEE.

Di Pietro, R. and Mancini, L. V. (2008). *Intrusion detection systems*, volume 38. Springer Science & Business Media.

Ding, S., Yang, S.-L., and Fu, C. (2012). A novel evidential reasoning based method for software trustworthiness evaluation under the uncertain and unreliable environment. *Expert Systems with Applications*, 39(3):2700–2709.

Disterer, G. (2013). Iso/iec 27000, 27001 and 27002 for information security management.

Dong, Y., Xu, Y., Li, H., and Dai, M. (2008). A comparative study of the numerical scales and the prioritization methods in ahp. *European Journal of Operational Research*, 186(1):229–242.

Dong, Y., Zhang, G., Hong, W.-C., and Xu, Y. (2010a). Consensus models for ahp group decision making under row geometric mean prioritization method. *Decision Support Systems*, 49(3):281–289.

Dong, Y., Zhang, G., Hong, W.-C., and Xu, Y. (2010b). Consensus models for ahp group decision making under row geometric mean prioritization method. *Decision Support Systems*, 49(3):281–289.

Dreiseitl, S. and Ohno-Machado, L. (2002). Logistic regression and artificial neural network classification models: a methodology review. In *Journal of Biomedical Informatics*, volume 35, pages 352–359.

Elia, I. A., Antunes, N., Laranjeiro, N., and Vieira, M. (2017). An analysis of openstack vulnerabilities. In *2017 13th European Dependable Computing Conference (EDCC)*, pages 129–134. IEEE.

Estabrooks, A., Jo, T., and Japkowicz, N. (2004). A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20(1):18–36.

Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51.

Feizi-Derakhshi, M.-R. and Ghaemi, M. (2014). Classifying different feature selection algorithms based on the search strategies. In *International conference on machine learning, electrical and mechanical engineering*, pages 17–21.

Feng, J., Xu, H., Mannor, S., and Yan, S. (2014). Robust logistic regression and classification. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

Fenton, N. and Pfleeger, S. (1997). Software metrics: A rigorous and practical approach.

Fernandez-Buglioni, E. (2013). *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons.

Gaffney Jr, J. E. (1981). Metrics in software quality assurance. In *Proceedings of the ACM'81 conference*, pages 126–130. ACM.

Galin, D. (2004). *Software quality assurance: from theory to implementation*. Pearson Education India.

Georganos, S. and et al. (2018). Very high resolution object-based land use–land cover urban classification using extreme gradient boosting. *IEEE Geoscience and Remote Sensing Letters*, pages 607–611.

Ghaffarian, S. M. and Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56.

Grefenstette, J. J. (2012). *Genetic Algorithms for Machine Learning*. Springer Science & Business Media.

Gupta, T. and Panda, S. P. (2019). A comparison of k-means clustering algorithm and clara clustering algorithm on iris dataset. *International Journal of Engineering & Technology*, 7(4).

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182.

Halfond, W. G. J., Choudhary, S. R., and Orso, A. (2011). Improving penetration testing through static and dynamic analysis. *Software Testing, Verification and Reliability*, 21(3):195–214.

Han, H., Guo, X., and Yu, H. (2016). Variable selection using mean decrease accuracy and mean decrease gini based on random forest. In *Software Engineering and Service Science (ICSESS), 2016 7th IEEE International Conference on*, pages 219–224. IEEE.

Hardin, R. (2002). Trust and trustworthiness.

Hasselbring, W. and Reussner, R. (2006). Toward trustworthy software systems. *Computer*, 39(4):91–92.

Heimann, D. (2014). Ieee standard 730-2014 software quality assurance processes. *IEEE Computer Society, New York, NY, USA, IEEE Std*, 730:2014.

Henrique Alves, Baldoino Fonseca, N. A. (2016). A dataset of source code metrics and vulnerabilities.

Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and del Cuvillo, J. (2013). Using innovative instructions to create trustworthy software solutions.

Horvath, A. S. and Agrawal, R. (2015). Trust in cloud computing. In *SoutheastCon 2015*, pages 1–8. IEEE.

Howard, M., Pincus, J., and Wing, J. (2005). Measuring relative attack surfaces. *Computer Security in the 21st Century*, pages 109–137.

Jayaswal, B. and Patton, P. C. (2006). Design for trustworthy software: Tools, techniques, and methodology of developing robust software.

Karim, S. and et al. (2017). 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. IEEE.

Kaufman, L. and Rousseeuw, P. J. (1990). Finding groups in data: An introduction to cluster analysis. *Wiley, New York*.

Kearney, J. P., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., and Adler, M. A. (1986). Software complexity measurement. *Communications of the ACM*, 29(11):1044–1050.

Ko, C., Ruschitzka, M., and Levitt, K. (1997). Execution monitoring of security critical programs in distributed systems: A specification-based approach. *Proceedings 1997 IEEE Symposium on Security and Privacy*, pages 175–187.

Kuhn, M. (2016). The r random forest package.

Kuhn, M., Wing, J., Weston, S., Williams, A., Keefer, C., Engelhardt, A., Cooper, T., Mayer, Z., Kenkel, B., Team, R. C., et al. (2020). Package 'caret'. *The R Journal*.

Lee, L. S. and Brink, W. D. (2020). Trust in cloud-based services: A framework for consumer adoption of software as a service. *Journal of Information Systems*, 34(2):65–85.

Li, Y. (2017). Software trustworthiness static measurement model and the tool. *International Journal of Performability Engineering*, 13.

Li, Z. and Shao, Y. (2019). A survey of feature selection for vulnerability prediction using feature-based machine learning. In *Proceedings of the 2019 11th International Conference on Machine Learning and Computing*, pages 36–42.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. (2018). Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756*.

Limam, N. and Boutaba, R. (2010). Assessing software service quality and trustworthiness at selection time. *IEEE Transactions on Software Engineering*, 36(4):559–574.

Liu, H., Dougherty, E. R., Dy, J. G., Torkkola, K., Tuv, E., Peng, H., Ding, C., Long, F., Berens, M., Parsons, L., et al. (2005). Evolving feature selection. *IEEE Intelligent systems*, 20(6):64–76.

Liu, M. Y. and Traore, I. (2006). Empirical relation between coupling and attackability in software systems: a case study on dos. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 57–64. ACM.

Lopes Margarido, I., Faria, J. P., Vidal, R. M., and Vieira, M. (2011). Classification of defect types in requirements specifications: Literature review, proposal and assessment. In *6th Iberian Conference on Information Systems and Technologies (CISTI 2011)*, pages 1–6.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. *In: Proceedings of the Symposium on Mathematics and Probability*, pages 281–297.

Marcil, J. (2014). Owasp iso iec 27034 application security controls project. *OWASP-Open Web Application Security Project*.

Martinez, M., de Andres, D., Ruiz, J.-C., and Friginal, J. (2014). From measures to conclusions using analytic hierarchy process in dependability benchmarking. *IEEE Transactions on Instrumentation and Measurement*, 63(11):2548–2556.

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, 2:80–83.

McGraw, G. (2006). *Software security: building security*, volume 1. Addison-Wesley Professional.

McGraw, G. and Viega, J. (2005). Building secure software. *Addition Wesley*.

Medeiros, N. and Basso, T. (2016). Perception of trustworthiness on web services and applications based on privacy evidences. In *7th Latin-American Symposium on Dependable Computing (LADC)*.

Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2017a). Software metrics as indicators of security vulnerabilities. In *28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227. IEEE.

Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2018a). An approach for trustworthiness benchmarking using software metrics. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 84–93. IEEE.

Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2018b). An approach for trustworthiness benchmarking using software metrics. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 84–93. IEEE.

Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2020). Vulnerable code detection using software metrics and machine learning. *IEEE Access*, 8:219174–219198.

Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2021). An empirical study on software metrics and machine learning to identify untrustworthy code. In *17th European Dependable Computing Conference (EDCC)*.

Medeiros, N., Ivaki, N. R., Costa, P. N. D., and Vieira, M. P. A. (2017b). Towards an approach for trustworthiness assessment of software as a service. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 220–223.

Medeiros, N., Ivaki, N. R., Costa, P. N. D., and Vieira, M. P. A. (2017c). Towards an approach for trustworthiness assessment of software as a service. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 220–223.

Medeiros, N., Ivaki, N. R., Costa, P. N. D., and Vieira, M. P. A. (2022). Trustworthiness models to categorize and prioritize code for security improvement. *Journal of Systems and Software (JSS), doi.org/10.1016/j.jss.2023.111621.*

Menzies, T., Greenwald, J., and Frank, A. (2006). Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering*, 33(1):2–13.

M.Graff and Wyk, V. (2003). Secure coding: principles and practices. *Designing and implementing secure applications*.

Mohammadi, N. G., Paulus, S., Bishr, M., Metzger, A., Könnecke, H., Hartenstein, S., Weyer, T., and Pohl, K. (2013). Trustworthiness attributes and metrics for engineering trusted internet-based software systems. In *International Conference on Cloud Computing and Services Science*, pages 19–35. Springer.

Mohammadi, N. G., Paulus, S., Bishr, M., Metzger, A., Könnecke, H., Hartenstein, S., Weyer, T., and Pohl, K. (2014). Trustworthiness attributes and metrics for engineering trusted internet-based software systems. *Communications in Computer and Information Science Cloud Computing and Services Science*, page 19–35.

Monga, M., Paleari, R., and Passerini, E. (2009). A hybrid analysis framework for detecting web application vulnerabilities. In *2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 25–32.

Moshtari, S., Sami, A., and Azimi, M. (2013). Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8–17.

Mudzingwa, D. and Agrawal, R. (2012). A study of methodologies used in intrusion detection and prevention systems (idps). In *2012 Proceedings of IEEE Southeastcon*, pages 1–6.

Myers, L. and Sirois, M. J. (2006). Spearman correlation coefficients, differences between. *Wiley StatsRef: Statistics Reference Online*.

Neto, A. and Vieira, M. (2011a). Selecting secure web applications using trustworthiness bemchmarking. *International Journal of Dependable and Trustworthy Information Systems*, 2.

Neto, A. and Vieira, M. (2011b). To benchmark or not to benchmark security: That is the question. *IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops*, page 182–187.

Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. (2007). Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM.

Nunes, P., Medeiros, I., Fonseca, J. C., Neves, N., Correia, M., and Vieira, M. (2018). Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175.

Ouffoué, G., Ortiz, A. M., Cavalli, A. R., Mallouli, W., Domingo-Ferrer, J., Sánchez, D., and Zaidi, F. (2016). Intrusion detection and attack tolerance for cloud environments: The clarus approach. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 61–66. IEEE.

Ouffoué, G., Zaïdi, F., and Cavalli, A. R. (2019). Attack tolerance for services-based applications in the cloud. In *IFIP International Conference on Testing Software and Systems*, pages 242–258. Springer.

Pan, J., Yan, G., and Fan, X. (2017). Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 149–165, Vancouver, BC. USENIX Association.

Patel, A., Qassim, Q., and Wills, C. (2010). Survey of intrusion detection and prevention systems. *Information Management and Computer Security*, 18.

Pham, D. and Karaboga, D. (2012). *Intelligent optimisation techniques: genetic algorithms, tabu search, simulated annealing and neural networks*. Springer Science & Business Media.

Potii, O., Illiashenko, O., and Komin, D. (2015). Advanced security assurance case based on iso/iec 15408. In *International Conference on Dependability and Complex Systems*, pages 391–401. Springer.

Poulin, L. and Guay, B. (2008). Iso/iec 27034 application security-overview. *20.1 Kyoto*, page 29.

Powers, D. M. W. (2020). Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation.

Rai, P. and Singh, S. (2010). A survey of clustering techniques. *International Journal of Computer Applications*, 7(12).

Rawat, M. S., Mittal, A., and Dubey, S. K. (2012). Survey on impact of software metrics on software quality. *IJACSA) International Journal of Advanced Computer Science and Applications*, 3(1).

Ren, J. and et al. (2019). A buffer overflow prediction approach based on software metrics and machine learning. *Security and Communication Networks*, 2019.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144.

Rish, I. et al. (2001). An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46.

Rosenberg, L., Hammer, T., and Shaw, J. (1998). Software metrics and reliability. In *9th international symposium on software reliability engineering*.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE.

S., S. and et. al. (1991). A survey of decision tree classifier methodology. In *IEEE Transactions on Systems, Man, and Cybernetics*.

Saket, S. and Pandya, S. (2016). An overview of partitioning algorithms in clustering techniques. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 5.

Saleh, K. and Elshahry, G. (2009). Modeling security requirements for trustworthy systems. *Encyclopedia of Information Science and Technology edited by Mehdi Khosrow-Pour*, pages 2657–2664.

Schapire, R. (2002). The boosting approach to machine learning -an overview. In *Nonlinear Estimation and Classification - Lecture Notes in Statistics*, volume 171, pages 149–171.

Schober, P. e. a. (2018). Correlation coefficients: Appropriate use and interpretation. *Anesthesia and analgesia*, 126(5):1763–1768.

SciTools (2017). Understand static code analysis tool.

security, I. J. S. . I. (2009). Iso 15408-1: 2009 - information technology - security techniques-evaluation criteria for it security. *cybersecurity and privacy protection*.

Sexton, R. S., Dorsey, R. E., and Johnson, J. D. (1999). Optimization of neural networks: A comparative analysis of the genetic algorithm and simulated annealing. *European Journal of Operational Research*, 114(3):589–601.

Shan, L., Sangchoolie, B., Folkesson, P., Vinter, J., Schoitsch, E., and Loiseaux, C. (2019). A survey on the applicability of safety, security and privacy standards in developing dependable systems. In *International Conference on Computer Safety, Reliability, and Security*, pages 74–86. Springer.

Shen, X. (2018). Predicting vulnerable files by using machine learning method. Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science(EWI), Delft University of Technology.

Shin, Y. (2008). Exploring complexity metrics as indicators of software vulnerability. In *Proc. of the Int. Doctoral Symp. on Empirical Soft. Eng.(IDoESE'08)*, page 3.

Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. (2010). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787.

Shin, Y. and Williams, L. (2008). Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 47–50. ACM.

Shin, Y. and Williams, L. (2011). An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pages 1–7. ACM.

Siavvas, M., Kehagias, D., and Tzovaras, D. e. a. (2021). A hierarchical model for quantifying software security based on static analysis alerts and software metrics. *Software Quality Journal*, page 431–507.

Sinaga, K. P. and Yang, M.-S. (2020). Unsupervised k-means clustering algorithm. *IEEE access*, 8:80716–80727.

Slemrod, J. and Katuscak, P. (2002). Do trust and trustworthiness pay off? Working Paper 9200, National Bureau of Economic Research.

Srinivas, M. and Patnaik, L. M. (1994). Genetic algorithms: A survey. *computer*, 27(6):17–26.

Tate, R. F. (1954). Correlation between a discrete and a continuous variable. point-biserial correlation. *The Annals of mathematical statistics*, 25(3):603–607.

Team, R. C. (2017). The r project for statistical computing.

Telang, R. and Wattal, S. (2007). An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software Engineering*, 33(8):544–557.

Turner, S. (2014). Security vulnerabilities of the top ten programming languages: C, java, c++, objective-c, c#, php, visual basic, python, perl, and ruby. *Journal of Technology Research*, 5:1.

Turpin, K. (2010). Owasp secure coding practices-quick reference guide.

Velasquez, M. and Hester, P. (2013). An analysis of multi-criteria decision making methods. *International Journal of Operations Research*, 10:56–66.

Vicente Mohino, J. d., Bermejo Higuera, J., Bermejo Higuera, J. R., and Sicilia Montalvo, J. A. (2019). The application of a new secure software development life cycle (s-sdlc) with agile methodologies. *Electronics*, 8(11).

Vieira, M., Antunes, N., and Madeira, H. (2009). Using web security scanners to detect vulnerabilities in web services. In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 566–571. IEEE.

Viera, A. J., Garrett, J. M., et al. (2005). Understanding interobserver agreement: the kappa statistic. *Fam Med*, 37(5):360–363.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 2000–02.

Walfish, S. (2006). A review of statistical outlier methods. *Pharmaceutical technology*, 30(11):82.

Walker, A., Coffey, M., Tisnovsky, P., and Černý, T. (2020). *On Limitations of Modern Static Analysis Tools*, pages 577–586.

Wang, A., An, N., Yang, J., Chen, G., Li, L., and Alterovitz, G. (2017). Wrapper-based gene selection with markov blanket. *Computers in biology and medicine*, 81:11–23.

Wang, H., Tang, Y., Yin, G., and Li, L. (2006). Trustworthiness of internet-based software. *Science in China Series F: Information Sciences*, 49:759–773.

Westreich, D., Lessler, J., and Funk, M. J. (2010). Propensity score estimation: neural networks, support vector machines, decision trees (cart), and meta-classifiers as alternatives to logistic regression. *Journal of clinical epidemiology*, 63(8):826–833.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Wood, J. M. (2007). Understanding and computing cohen's kappa: A tutorial. *WebPsychEmpiricist. Web Journal at http://wpe.info/.*

Wysopal, C. and et. al (2006). *The Art of Software Security Testing: Identifying Software Security Flaws*. Pearson Education.

Yang, Y., Wang, Q., and Li, M. (2009). Process trustworthiness as a capability indicator for measuring and improving software trustworthiness. volume 5543, pages 389–401.

Yu, C. H. (2002). Resampling methods: concepts, applications, and justification. *Practical Assessment, Research, and Evaluation*, 8(1):19.

Yu, L. and Liu, H. (2004). Efficient feature selection via analysis of relevance and redundancy. *Journal of machine learning research*, 5(Oct):1205–1224.

Zhao, J. and Gong, R. (2015). A new framework of security vulnerabilities detection in php web application. In *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 271–276.

Appendixes

A - Software Metrics

This Appendix presents an extended list of software metrics from different types: complexity, volume, coupling and cohesion. The metrics used in the experimental evaluations throughout this thesis are identified in blue.

Table A.1: Extended list of complexity metrics.

Metrics Short Name	Metrics Name	Description
AvgCyclomatic	Average Cyclomatic Complexity	Average cyclomatic complexity for all nested functions or methods
AvgEssential	Average Essential Cyclomatic Complexity	Average Essential complexity for all nested functions or methods
AvgEssentialStrictModified	Average Essential Strict Modified Complexity	Average strict modified essential complexity for all nested functions or methods
AvgCyclomaticModified	Average Modified Cyclomatic Complexity	Average modified cyclomatic complexity for all nested functions or methods
AvgMaxNesting	Average Nesting	Average of maximum nesting level of control constructs
AvgCyclomaticStrict	Average Strict Cyclomatic Complexity	Average strict cyclomatic complexity for all nested functions or methods
RatioCommentToCode	Comment to Code Ratio	Ratio of comment lines to code lines
Cyclomatic	Cyclomatic Complexity	Cyclomatic complexity
MaxInheritanceTree	Depth of Inheritance Tree	Maximum depth of class in inheritance tree
Essential	Essential Complexity	Essential complexity
EssentialStrictModified	Essential Strict Modified Complexity	Strict Modified Essential complexity
HK	Henry Kafura	Measures information flow relative to function size
Knots	Knots	Measure of overlapping jumps
MaxCyclomatic	Max Cyclomatic Complexity	Maximum cyclomatic complexity of all nested functions or methods
MaxEssential	Max Essential Complexity	Maximum essential complexity of all nested functions or methods
MaxEssentialStrictModified	Max Essential Strict Modified Complexity	Maximum strict modified essential complexity of all nested functions or methods
MaxEssentialKnots	Max Knots	Maximum Knots after structured programming constructs have been removed
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
MaxMaxNesting	Maximum Nesting	Maximum nesting level of control construct
MinEssentialKnots	Minimum Knots	Minimum Knots after structured programming constructs have been removed
CyclomaticModified	Modified Cyclomatic Complexity	Modified cyclomatic complexity
MaxNesting	Nesting	Nesting level of control constructs
CountPath	Paths	Number of possible paths, not counting abnormal exits
CyclomaticStrict	Strict Cyclomatic Complexity	Strict cyclomatic complexity
SumCyclomatic	Sum Cyclomatic Complexity	Sum of cyclomatic complexity of all nested functions or methods
SumEssential	Sum Essential Complexity	Sum of essential complexity of all nested functions or methods
SumEssentialStrictModified	Sum Essential Strict Modified Complexity	Sum of strict modified essential complexity of all nested functions or methods
SumCyclomaticModified	Sum Modified Cyclomatic Complexity	Sum of modified cyclomatic complexity of all nested functions or methods
SumMaxNesting	Sum nesting	Sum of maximum nesting level of control constructs
SumCyclomaticStrict	Sum Strict Cyclomatic Complexity	Sum of strict cyclomatic complexity of all nested functions or methods

Table A.2: Extended list of volume metrics.

Metrics Short Name	Metrics Name	Description
AltAvgLineBlank	Average Number of Blank Lines	Average number of blank lines for all nested functions, including inactive regions
AltAvgLineCode	Average Number of Lines of Code	Average number of lines containing source code for all nested functions, including inactive regions
AltAvgLineComment	Average Number of Lines with Comments	Average number of lines containing comment for all nested functions,including inactive regions
AltCountLineBlank	Blank Lines of Code	Number of blank lines, including inactive regions
AltCountLineCode	Lines of Code	Number of lines containing source code, including inactive regions
AltCountLineComment	Lines with Comments	Number of lines containing comment, including inactive regions
AvgLine	Average Number of Lines	Average number of lines for all nested functions
AvgLineBlank	Average Number of Blank Lines	Average number of blank for all nested functions
AvgLineCode	Average Number of Lines of Code	Average number of lines containing source code for all nested functions
AvgLineComment	Average Number of Lines with Comments	Average number of lines containing comment for all nested functions
CountDeclFile	Number of Files	Number of files
CountDeclFunction	Function	Number of functions
CountDeclInstanceVariableInternal	Internal Instance Variables	Number of internal instance variables
CountDeclInstanceVariableProtectedInternal	Protected Internal Instance Variables	Number of protected internal instance variables
CountDeclMethodProtectedInternal	Local Protected Internal Methods	Number of local protected internal methods
CountDeclMethodFriend	Friend Methods	Number of local friend methods
CountDeclMethodInternal	Local Internal Methods	Number of local internal methods
CountInput	Inputs	Number of calling subprograms plus global variables read (FanIn)
CountLine	Physical Lines	Number of all lines
CountLineBlank	Blank Lines of Code	Number of blank lines
CountLineCode	Source Lines of Code	Number of lines containing source code
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
CountLineInactive	Inactive Lines	Number of inactive lines
CountLinePreprocessor	Preprocessor Lines	Number of preprocessor lines
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

Table A.3: Extended list of coupling and cohesion metrics.

Metrics Short Name	Metrics Name	Description
AvgFanIn	Average Input	Average number of calling subprograms plus global variables read
AvgFanOut	Average Output	Average number of called subprograms plus global variables set
CountClassCoupled	Coupling Between Objects	Number of other classes coupled to [CBO (coupling between object classes)]
CountClassDerived	Number of Children	Number of immediate subclasses [NOC (number of children)]
CountDeclClass	Classes	Number of classes
CountDeclClassMethod	Class Methods	Number of class methods
CountDeclClassVariable	Class Variables	Number of class variables
CountDeclFunction	Function	Number of functions
CountDeclInstance VariablePublic	Public Instance Variables	Number of public instance variables
CountDeclInstance VariablePrivate	Private Instance Variables	Number of private instance variables
CountDeclInstance VariableProtected	Protected Instance Variables	Number of protected instance variables
CountDeclInstanceMethod	Instance Methods	Number of instance methods
CountDeclInstanceVariable	Instance Variables	Number of instance variables
CountDeclMethod	Local Methods	Number of local methods
CountDeclMethod StrictPublished	Local strict published methods	Number of local strict published methods
CountDeclMethodAll	Methods	Number of methods, including inherited ones
CountDeclMethodConst	Local Const Methods	Number of local const methods
CountDeclMethodDefault	Local Default Visibility Methods	Number of local default 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
CountDeclMethodStrictPrivate	Local strict private methods	Number of local strict private methods
CountDeclModule	Modules	Number of modules
CountDeclProgUnit	Program Units	Number of non-nested modules, block data units, and subprograms
CountDeclSubprogram	Subprograms	Number of subprograms.
CountInput	Inputs	Number of calling subprograms plus global variables set [FanIn]
CountOutput	Outputs	Number of called subprograms plus global variables set [FanOut]
CountPackageCoupled	Coupled Packages	Number of other packages coupled to
MaxFanIn	Maximum Input	Maximum number of calling subprograms plus global variables set
MaxFanOut	Maximum Output	Maximum number of called subprograms plus global variables set
MaxInheritanceTree	Depth of Inheritance Tree	Maximum depth of class in inheritance tree [DIT]
PercentLackOfCohesion	Lack of Cohesion in Methods	100% minus the average cohesion for package entities [LCOM]