UNIVERSIDADE D
COIMBRA

João Miguel Namorado Clímaco Henggeler Antunes

BUILDING AND EVALUATING SOFTWARE VULNERABILITY DATASETS

Combining Static Analysis Alerts and Software Metrics to Automatically Detect Vulnerable C/C++ Functions

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professors José D'Abruzzo Pereira and Marco Vieira, and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Building and Evaluating Software Vulnerability Datasets

Combining Static Analysis Alerts and Software Metrics to Automatically Detect Vulnerable C/C++ Functions

João Miguel Namorado Clímaco Henggeler Antunes

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Professors José D'Abruzzo Pereira and Marco Vieira, and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

This work is within the informatics security specialization area and was carried out in the Software and Systems Engineering (SSE) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work has been supported by the project **AIDA** - Adaptive, Intelligent and Distributed Assurance Platform (reference `POCI-01-0247-FEDER-045907`) leading to this work is co-financed by the ERDF and COMPETE 2020 and by the FCT under CMU Portugal.

This page is intentionally left blank.

Acknowledgements

I would like to thank my advisors, José D'Abruzzo Pereira and Marco Vieira, for their guidance, insight, and availability throughout this entire work's development.

To my family and friends, for their unconditional love and support during the COVID-19 pandemic.

This page is intentionally left blank.

Abstract

Software vulnerabilities can have serious consequences when exploited, such as unauthorized authentication, data breaches, and financial losses. Manually reviewing an entire codebase for weaknesses is cumbersome, time-consuming, and sometimes impossible depending on a project's size. Due to the nature of this industry, companies are increasingly pressured to deploy and update software as quickly as possible. Automated tools called Static Analysis Tools (SATs) can generate security alerts that highlight potential vulnerabilities in an application's source code, though they are prone to misidentified vulnerabilities called false positives.

In this work, we present an automated process capable of collecting new vulnerabilities from the CVE Details website, retrieving affected files, functions, and classes from a project's repository, generating software metrics and security alerts (i.e. potential vulnerabilities), and building robust datasets capable of being fed to machine learning algorithms. We put this mechanism into practice by creating vulnerable code unit datasets for five large and widely known C/C++ projects: Mozilla, Linux Kernel, Xen Hypervisor, Apache HTTP Server, and GNU C Library.

Additionally, the created vulnerable function dataset is validated using a wide assortment of machine learning parameters, so as to build and find the best classifiers capable of labeling functions as vulnerable, neutral, or belonging to a specific vulnerability category. Results show that it is possible to use both software metrics and security alerts to detect vulnerable function code, with precision, recall, and F-score values as high as 93.7%, 95.1%, and 93.9%, respectively. Moreover, further analysis into the influence of a vulnerability's detection year on the classifiers' performance was carried out. However, it could not be determined if using static data from previous years could be used to detect vulnerable functions in later ones.

Keywords

Software Security, Vulnerability Detection, Static Code Analysis, Software Metrics, Machine Learning

This page is intentionally left blank.

Resumo

As vulnerabilidades de software podem ter consequências graves caso sejam exploradas, incluindo acessos não autorizados, violações de dados, e perdas financeiras. O processo de rever código manualmente é tanto complexo como demorado, sendo por vezes inviável de aplicar dependendo do tamanho de um projeto. Por outro lado, as empresas de software são cada vez mais encorajadas a publicar e atualizar os seus produtos o mais rapidamente possível. Apesar de existirem ferramentas que encontram potenciais vulnerabilidades automaticamente no código fonte, estas geraram um número elevado de falsos positivos, ou vulnerabilidades mal classificadas. Para além disso, este tipo de técnicas nem sempre são suficientemente fiáveis para detetar vulnerabilidades.

No presente trabalho apresentamos um processo automatizado capaz de recolher novas vulnerabilidades a partir do website CVE Details, selecionar ficheiros, funções, e classes afetadas do repositório de cada projeto, gerar métricas de software e alertas de segurança (i.e. potenciais vulnerabilidades), e construir datasets robustos de modo a serem processados por algoritmos de aprendizagem computacional. Este mecanismo foi usado para desenvolver datasets de unidades de código vulneráveis para cinco projetos implementados em C/C++: Mozilla, Linux Kernel, Xen Hypervisor, Apache HTTP Server, e GNU C Library.

Adicionalmente, o dataset relativo a funções vulneráveis foi validado através de modelos de aprendizagem computacional, de modo a determinar quais os parâmetros que geravam os melhores classificadores. Os resultados experimentais demonstram que é possível usar tanto métricas de software como alertas de segurança para detetar funções vulneráveis, tendo sido obtidos valores de precisão, revocação, e F-score de 93.7%, 95.1%, e 93.9%, respetivamente. Foi também feita uma análise sobre a influência do ano em que as vulnerabilidades foram descobertas no desempenho destes classificadores. No entanto, não foi possível determinar se o uso de dados de anos anteriores permite a deteção de funções vulneráveis nos anos seguintes.

Palavras-Chave

Segurança de Software, Deteção de Vulnerabilidades, Análise Estática de Código, Métricas de Software, Aprendizagem Computacional

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

AME Alhazmi-Malaiya Effort-Based Model. 9

AML Alhazmi-Malaiya Logistic Model. 8

AST Abstract Syntax Tree. 29, 47

AT Anderson Thermodynamic Model. 8

CBO Coupling Between Objects. 13, 20, 60, 64

CCC Complexity, Coupling, and Cohesion. 13

CK Chidamber and Kemerer. 13, 16, 17, 31

CSV Comma-Separated Values. xvi, 28, 30, 49, 62–67, 71, 72

CVE Common Vulnerabilities and Exposure. xix, 1, 3, 19, 25–27, 30, 32–35, 40, 41, 62

CVSS Common Vulnerability Scoring System. 25, 32, 62

CWE Common Weakness Enumeration. xvi, xix, 18, 21, 23, 25, 31–36, 62–64, 68

DBMS Database Management System. 32

DIT Depth of Inheritance Tree. 13, 60, 64

DLL Dynamic Link Library. 62, 72

ER Entity–Relationship. xvi, 32–35

FANIN Fan-In. 14–16, 31, 36, 60

FANOUT Fan-Out. 14–16, 20, 31, 36, 60

FN False Negative. 39

FP False Positive. 39

HK Henry Kafura Size. 14, 15, 20, 60

HTTP Hypertext Transfer Protocol. 68

IDE Integrated Development Environment. 42

IEEE Institute of Electrical and Electronics Engineers. 1

JSON JavaScript Object Notation. 67, 71, 73, 79

LCOM Lack of Cohesion in Methods. 14, 20, 60, 64

LOC Lines of Code. 12, 14–16, 20, 31, 36, 60

LP Logarithmic Poisson Model. 9

MFSA Mozilla Foundation Security Advisories. 19

ML Machine Learning. xxii

NIST National Institute of Standards and Technology. 1

NOC Number of Children. 13, 60, 64

NVD National Vulnerability Database. 1, 17

OWASP Open Web Application Security Project. 21

QMOOD Quality Model for Object-Oriented Design. 17

RE Rescorla Exponential Model. 9

RF Random Forest. 39, 42–44, 52, 66

RFC Response for a Class. 13, 60, 64

RL Rescorla Linear Model. 9

ROC Receiver Operating Characteristic. 16, 17

RQ Research Question. 37, 38, 40, 41, 44, 47, 51, 52

SAT Static Analysis Tool. vii, xvi, xix, xxii, 2, 3, 7, 15–18, 21–23, 30–36, 38, 41, 44, 47, 51, 52, 62, 65, 67, 68, 72–74

SCA Static Code Analysis. 2, 3, 7, 12, 18, 21–24, 26, 28–30, 51, 67, 68

SLOC Source Lines of Code. 60

SM Software Metric. xxii

SMOTE Synthetic Minority Oversampling Technique. 20

SQL Structured Query Language. 63, 67, 72

SQLi SQL Injection. 16

SVM Support Vector Machine. 17, 18, 20, 47

TN True Negative. 39

TP True Positive. 39

URL Uniform Resource Locator. 26, 32, 62, 67

VDM Vulnerability Discovery Model. xvi, 8–10

WMC Weighted Methods Per Class. 13, 60

XGB Extreme Gradient Boosting. 39, 44

XSA Xen Security Advisories. xvi, 19, 27

XSS Cross-Site Scripting. 16, 17

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

List of Publications

This dissertation contributed to the following publication which was submitted and accepted by a workshop:

- José D'Abruzzo Pereira, **João Henggeler Antunes**, and Marco Vieira. On building a vulnerability dataset with static information from the source code. In *Safety, Security, and Privacy in Complex Artificial Intelligence based Systems (SAFELIFE 2021)*, 2021. (accepted).

  - **Abstract:** Software vulnerabilities are weaknesses in software systems that can have serious consequences when exploited. Examples of side effects include unauthorized authentication, data breaches, and financial losses. Due to the nature of the software industry, companies are increasingly pressured to deploy software as quickly as possible, leading to a large number of undetected software vulnerabilities. Static code analysis, with the support of SATs, can generate security alerts that highlight potential vulnerabilities in an application's source code. Software Metrics (SMs) have also been used to predict software vulnerabilities, usually with the support of Machine Learning (ML) classification algorithms. Several datasets are available to support the development of improved software vulnerability detection techniques. However, they suffer from the same issues: they are either outdated or use a single type of information. In this paper, we present a methodology for collecting software vulnerabilities from known vulnerability databases and enhancing them with static information (namely SAT alerts and SMs). The proposed methodology aims to define a mechanism capable of more easily updating the collected data.

This page is intentionally left blank.

Chapter 1

Introduction

Securing one's information from foreign entities only becomes more important with time. According to a 2020 technical report by IBM Security [44], data breaches averaged a total of $3.86 million worldwide, with more than half being caused by malicious user attacks. More worrying still, the healthcare industry accounted for the highest average cost of $7.13 million, just as the COVID-19 pandemic spread around the globe. This may only become exacerbated with time as the majority of companies predicted that remote work, a staple of the pandemic, would increase both the cost and response time of dealing with a data breach.

Software systems are prone to bugs, or accidental faults which can themselves lead to errors [19, 37]. According to the Institute of Electrical and Electronics Engineers (IEEE), these errors range from following a wrong set of instructions to producing an unexpected outcome during human interaction [17]. Software vulnerabilities are a subset of bugs that can be maliciously used to compromise a system and undermine its security. This can be accomplished, for example, by running an application and performing certain steps in a given order, by changing its environment, or by taking advantage of a system's inherently insecure behavior. The act of using vulnerabilities for these malicious ends is called exploiting [19].

The CVE system, a widely used list of publicly known security vulnerabilities, also defines software vulnerabilities as "a flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components" [18]. Likewise, the National Vulnerability Database (NVD), a vulnerability database that collates information from CVE and that is managed by the National Institute of Standards and Technology (NIST), borrows this same definition for their operations [39].

In addition to the loss of sensitive user data and disruption of service, software vulnerabilities may lead to a significant financial impact. A 2016 security report published by the Kaspersky Labs [29] found that zero-day vulnerabilities (previously unknown vulnerabilities that have yet to be patched, meaning they can be exploited by malicious users) were the most expensive security incidents for both enterprises and small and medium-sized businesses. While rare, these vulnerabilities cost an estimated $149 thousand for small and medium-sized businesses and $2 million for enterprises. Small and medium-sized businesses also ranked vulnerabilities in in-house software as the sixth most expensive incident and known unpatched exploits in off-the-shelf software and hardware as the eighth. The amount was $115,000 and $101,000 in financial losses, respectively.

Much like in other industries, the announcement of defects has a negative influence on a software vendor's market value. Telang and Wattal [48] found that vendors lose on average 0.63% of their value, or an estimated $0.86 billion loss, on the day that the vulnerability is announced. This effect is worsened by other factors, including the market's competition, a small vendor size, the vulnerability's severity, and the time it takes to patch it. A vendor that does not provide a patch when the vulnerability is disclosed to the public suffers a 0.82% greater loss than others that did so. Vulnerabilities that violate confidentiality were found to also result in greater losses. As a whole, this cost comes from having to patch the vulnerability, and from the loss of a vendor's reputation and future sales.

By successfully exploiting a vulnerability one may potentially affect the following key information security properties [19]:

- **Confidentiality**: the act of keeping information private and preventing any unauthorized access. Software systems often store and interface with sensitive user data like personal files, credentials, or financial information. Failing to fulfill this attribute could allow third parties to use the data for unlawful purposes. It is expected that the system is able to maintain a high level of privacy for the end-user, regardless of who they are;
- **Integrity**: the act of ensuring the correctness of data and preventing any unwanted modifications to their source or contents. This may be accomplished by stopping any unauthorized changes to the former and establishing integrity detection mechanisms for the latter. Should this property be circumvented, third parties may impersonate a user or tamper with their information;
- **Availability**: the act providing uninterrupted access to the system and preventing any attacks that could affect its service. Not doing so would violate a user's expectations, and would render the system's functionalities and resources inaccessible.

As software becomes more complex and interconnected, so too does the difficulty and cost in creating secure systems increase [22]. Moreover, developers often have to maintain legacy codebases, and cannot afford to rebuild them from the ground up [19]. Application security thus becomes a necessity and, for big companies with equally large software codebases, there must exist mechanisms to detect any vulnerabilities that they might contain.

One way of doing this is by using what are called software vulnerability discovery techniques, which divide themselves into two categories depending on how they perform their analysis: static techniques that merely check an application's source files, and dynamic techniques that execute a program in a controlled environment [32]. A traditional example of Static Code Analysis (SCA) would be code inspections, where people review code manually by referencing predefined guidelines [20]. Likewise, a related dynamic approach would be test cases, where a set of inputs and steps are fed into a system, in hopes that an expected output is produced. In this first category sits static analysis, a technique that can be particularly useful for detecting weaknesses in large projects as it is fast, scalable, and easy to integrate into their development process. Unlike manual SCA, this technique can be automated using what are called SATs. These tools highlight potential vulnerabilities in specific code locations by generating security alerts. This type of approach, however, is subject to a high number of misidentified vulnerabilities, called false positives. As such, relying solely on static analysis to find vulnerabilities may not be advisable [40].

In a similar vein are constructs called software metrics, which describe static code properties that can be used to gauge a software project's quality during its life cycle [38]. These

metrics have been used successfully in the past to distinguish between vulnerable and non-vulnerable code, though there is no single metric that applies to all cases [35]. More specifically, other studies have combined and investigated the usage of software metrics with machine learning algorithms in order to detect vulnerabilities [6, 47].

1.1 Contributions

Alves et al. [5] created a database of vulnerabilities from five large C/C++ projects, where each affected file, function, and class is associated with an assortment of software metrics collected using a SAT. However, this information only pertains to the years 2000 until 2016, meaning it is now out of date. As such, we developed an automated process capable of collecting new vulnerabilities from online platforms, retrieving each affected file from a project's version control system, generating software metrics using the same tools, and updating the original database. Moreover, this data is enriched with security alerts, which represent potential vulnerabilities, also generated via SCA.

Having collected this new information, we developed three datasets composed of software metrics and security alert occurrences in each project's files, functions, and classes. These were made specifically to be ingested by machine learning algorithms, so as to create classifiers capable of labeling unseen code units as vulnerable or not. Because we complemented the database with each vulnerability's category, we are also able to classify these code units as belonging to specific categories.

Indeed, we trained and tested machine learning models using a wide array of parameters in order to assign functions a binary label (vulnerable or not), or a multiclass label (vulnerable with a given category). By exploring different configurations, we were able to find the parameters that yielded the best results along the well-known precision, recall, and F-score performance metrics [50] for both labels. Results showed that it is possible to use both software metrics and security alerts to identify vulnerable functions, with precision, recall, and F-score values as high as 93.7%, 95.1%, and 93.9%, respectively.

We took this opportunity to analyze the influence of certain properties in vulnerable functions and chose to focus on a vulnerability's discovery date. Since the year a vulnerability was published is encoded in the CVE identifier, we were able to divide each function into groups from 2002 to 2019. By using a temporal sliding window, which considered functions in a given year range for the training subset and the year immediately after as the testing data, we checked whether this time component could influence the detection of vulnerable function code. Although some results showed values as high as the previous exploration stage, there was also a significant drop in performance for the years between 2017 and 2019. We also found that different sliding window sizes resulted in very similar values for the same performance metric. We reached the conclusion that the data partitioning scheme likely led to a worse testing subset in terms of size and representation. Thus, it was not possible to determine whether using static data from earlier vulnerability years could be used to predict if a function was vulnerable or not.

1.2 Document Outline

The rest of this document is organized as follows: Chapter 2 introduces some software vulnerability discovery techniques as well as various traditional software metrics. This chapter also showcases other related studies by describing their methodologies and results.

In Chapter 3, we provide an in-depth explanation of how each code unit dataset was built, from collecting vulnerabilities from online platforms to preparing the data so it could be fed to machine learning models. Chapter 4 describes how the function dataset was validated in practice. This includes listing which classification algorithms, data processing techniques, and hyperparameters were used, as well as presenting the performance metrics obtained after running the exploratory and year-based experiments. Any threats to validity that may have hindered our work are also shown here. Next, Chapter 5 describes the work's environment, including how its files and tasks were organized between the student and advisors, and how some important implementation problems were solved. Lastly, we present our main conclusions and any future work in Chapter 6.

This page is intentionally left blank.

Chapter 2

State of the Art

This chapter introduces vulnerability discovery techniques, including static techniques (like static analysis and vulnerability discovery models) and dynamic techniques (like dynamic analysis, penetration testing, and fuzzing). Software metrics that can be collected from procedural and object-oriented code using static techniques are also described. Finally, we overview existing studies and articles that are related to our work.

One way of detecting vulnerabilities is by using software vulnerability discovery techniques. These focus on discovering undetected vulnerabilities where a key principle is that, the longer it takes to find one, the more expensive it is to fix it [27]. The diagram in Figure 2.1 shows a graphical representation of the vulnerability discovery process using some of the concepts introduced in Chapter 1. Here, one may see how the previously mentioned errors can compromise a system by opening it to outside malicious users.

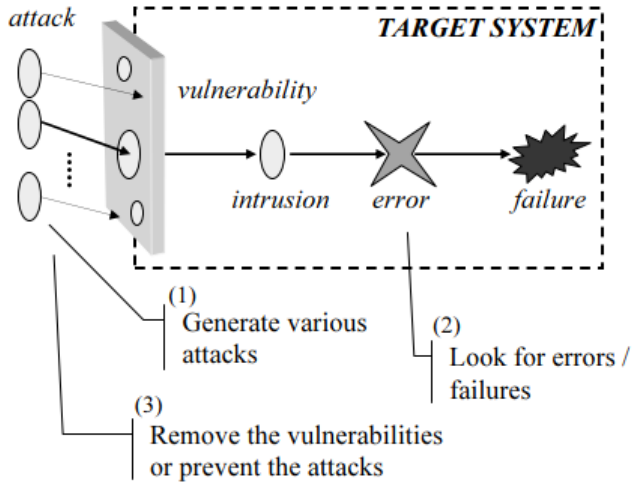


Figure 2.1: A diagram of the vulnerability discovery methodology. Taken from [37].

Some of these detection methods are described in the next few subsections, including static analysis, vulnerability discovery models, dynamic analysis, penetration testing, and fuzzing. As we will see, these approaches fall under one of two categories: static techniques and dynamic techniques.

2.1 Static Vulnerability Discovery Techniques

In this type of vulnerability discovery technique, an application's vulnerabilities and weaknesses are assessed without running it. Below, two specific static techniques are presented: static analysis and vulnerability discovery models [32].

2.1.1 Static Analysis

Static analysis is a process where the behavior of an application is analyzed, without executing it, in order to discover any defects or weaknesses [13]. One traditional example is manual code inspection, where source code is reviewed by people and certain predetermined properties are verified [20]. Since methods like these are time-consuming and require prior knowledge of the vulnerabilities to detect, automated tools are created to aid Static Code Analysis (SCA) [32]. These are called Static Analysis Tools (SATs), though they are not fully automated as their output must be verified by a human.

Because running the program is not required, the main advantages of this type of technique are its fast execution time and ease of integration into the software development cycle [32]. Additionally, this method is considered sound, i.e., its analysis holds for all program executions. It does, however, have some drawbacks, such as a high number of false positives (misidentified vulnerabilities), being prone to false negatives (undetected vulnerabilities), and requiring human validation at some level. Moreover, it is unable to detect design flaws or vulnerabilities caused by software configurations.

More specific static analysis techniques also exist, including:

- **Pattern Matching**: a process where a potentially dangerous textual pattern is searched for in the source code [27]. For example, locating a function call that is inherently unsafe and whose usage is no longer recommended. Due to its simplicity, this technique is very limited and generates a high number of false positives;
- **Lexical Analysis**: a process where a program's source code is subdivided into a set of identifiers with a given meaning in its language (tokens), which are then compared to a database of known defects [27, 31]. This process also suffers from a high number of false positives given that it doesn't take the program's syntax or semantics into account;
- **Type Inference**: a process where the data types of variables and functions are inferred and checked to see if they're in accordance with the rules defined in the programming language [31];
- **Syntactic Pattern Matching**: a process where an application is broken down into a structure called the abstract syntax tree, which represents the program's constructs and rules [23]. The analysis is then conducted by searching for a set of potentially dangerous operations in this structure. This static analysis technique is one of the fastest, but can also generate many false positives while providing little insight into a program's correctness;
- **Data Flow Analysis**: a process where a program is converted into a graph, called the control flow graph, that represents every path that may be traversed during its execution [23, 31]. Each node shows a sequence of instructions, where any connecting edges represent a branch in execution, allowing us to determine the possible values a variable or expression can take during its lifetime. This process can be used for

several purposes like finding the most recent assignment to a variable, removing unused assignments, removing redundant operations, or optimizing reusable arithmetic results. One special use case of this technique is taint analysis, where untrusted data is monitored as it propagates through the graph's nodes [27]. This may be used to check if any potentially dangerous input ever reaches a critical part of the application without being properly transformed into trustworthy data (sanitized);

- **Constraint-Based Analysis**: a process in which a set of constraints that describe the program's information and behavior are created [23, 31]. This system of constraints is then solved, and any relevant information is extracted from its solution;
- **Symbolic Execution**: a process where the program's input is converted to symbols that represent arbitrary values rather than actual data [23, 31]. These are associated with algebraic expressions, meaning the application's output values are now computed as a function of these formulas. By applying constraints to these input symbols, it's possible to build a tree structure that represents the program's execution paths. Constraint solvers can then be used to determine which set of inputs result in certain execution paths, potentially detecting errors. This method is prone to scalability problems as a large number of execution states can exceed the constraint solvers' problem-solving abilities. Moreover, issues with consistency may arise if the program interacts with components outside its environment [30];
- **Theorem Proving**: a process in which the program is converted into logic formulas based on its preconditions and post-conditions [23, 31]. The validity of this program theorem is then proved by using axioms and inference rules.

2.1.2 Vulnerability Discovery Models

A Vulnerability Discovery Model (VDM) shows the evolution of the number of detected vulnerabilities in a given software during its life cycle, allowing end-users to assess its security risks [32]. These models attempt to best reflect reality by estimating the rate of vulnerability discovery and the cumulative number of vulnerabilities discovered. As such, their purpose is to evaluate a software's security as a whole, instead of detecting specific vulnerable components [33]. Their results are then subject to statistical tests that examine how well the models tracked the discovery process. VDMs can be categorized as time-based or effort-based given their domain. Time-based VDMs are parametric functions that estimate the total number of vulnerabilities at a particular instance since the software was released, whereas effort-based VDMs base this value on the software's testing effort. Below, several VDMs are described [3]:

- **Anderson Thermodynamic Model (AT)**: a time-based model that assumes that any detected vulnerabilities are removed and that no more are reintroduced in the process. The cumulative number of vulnerabilities grows as a logarithmic function. One of its parameters represents the lower failure rate of the software's user beta testing phase as compared with alpha testing;
- **Alhazmi-Malaiya Logistic Model (AML)**: a time-based model that bases itself on three stages of an operating system's usage during its life cycle: 1) the learning phase, where the attention to the newly released software grows slowly, resulting in few reported vulnerabilities; 2) the linear phase, where the number of reported vulnerabilities increases as more and more people become familiar with the software; 3) the saturation phase, where the number of detected vulnerabilities starts declining

due to decreased user interest and a reduced vulnerability pool. The evolution of the cumulative number of vulnerabilities is represented as an S-shaped (sigmoid) curve;

- **Alhazmi-Malaiya Effort-Based Model (AME)**: an effort-based VDM that estimates its value based on the number of users in a system instead of relying on calendar time [4]. The reasoning behind this model is that a greater effort for discovering vulnerabilities, both inside and outside an organization, would go towards a system with a large userbase;
- **Rescorla Linear Model (RL)**: a time-based model that attempts to estimate the vulnerability detection rate linearly with time, rather than using the cumulative number of vulnerabilities. As time increases, this cumulative amount grows polynomially. Another similar model is the **Rescorla Exponential Model (RE)** which presents an exponential growth where this cumulative value approaches the total number of vulnerabilities in the system as time increases;
- **Logarithmic Poisson Model (LP)**: a time-based model where the cumulative number of vulnerabilities increases indefinitely with a logarithmic growth. In this case, finding a physical interpretation for both the model and its parameters is complex. Despite this, it can present better results than other models in many cases.

The graph in Figure 2.2 plots different time-based VDMs whose parameters have been specified so that their curve best approximates the real number of detected vulnerabilities in the Windows 95 operating system.

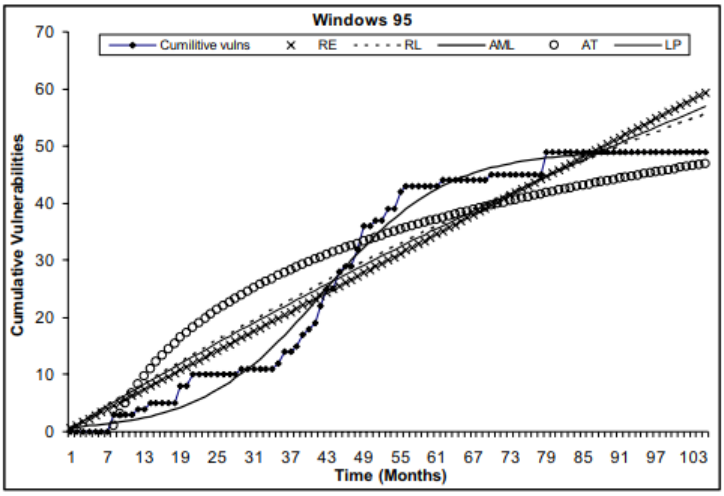


Figure 2.2: The cumulative number of detected vulnerabilities in Windows 95 over time and various time-based VDMs that attempt to fit this data. Taken from [3].

This technique is advantageous as it can theoretically predict both the rate of vulnerability discovery and the total amount of vulnerabilities, allowing organizations to identify current trends and make future projections of a given software's life cycle [32]. Their prediction capabilities can also be improved if more information is known about a system's environment since their parameters can be constrained to specific values. As models, however, they can only approximate reality, meaning their performance could be poor if certain factors aren't properly accounted for. It should also be noted that VDMs ignore a system's architecture and assume that vulnerabilities can be found in any of its components. All vulnerabilities are treated equally, regardless of how severe each one may be. Moreover, VDMs assume that different software releases are independent, while in practice a significant amount of

components are reused [33]. Using a single VDM may also lack generality for a system's entire lifespan, requiring more than one model to describe the same software at different periods in time.

2.2 Dynamic Vulnerability Discovery Techniques

In this type of vulnerability discovery technique, the analysis is conducted by executing and interfacing with software in a controlled environment. Below, three specific dynamic techniques are presented: dynamic analysis, penetration testing, and fuzzing [32].

2.2.1 Dynamic Analysis

Dynamic analysis is a process in which a program is analyzed by running it inside a controlled environment and monitoring its behavior [30]. In this approach, defects are found by producing test cases that allow the analyzers to cover as many different execution paths as possible. Since it is performed in a running application, vulnerability detection is generally more accurate and generates fewer false positives than static analysis [1].

Despite this, however, dynamic analysis may require a high level of human involvement with strong technical skills. As such, this process can have poor scalability and result in slower vulnerability detection. Due to their nature, the discovery process also suffers from runtime overhead. Moreover, defects present in unexecuted paths are not found, requiring a large number of test cases to ensure a certain confidence level in detecting vulnerabilities. No conclusions can then be made about code that wasn't executed during the analysis [43]. Because of this, some methodologies combine both static and dynamic analysis to improve code coverage and detection speed while reducing the number of false positives [1, 51]. Like static analysis, this technique also supports taint analysis, where untrustworthy data is tracked throughout the program's execution in order to verify if it reaches any critical components [51]. However, this method still suffers from the previous drawbacks, including not being able to detect latent weak spots.

2.2.2 Penetration Testing

Penetration testing is a method for evaluating the security of a system by simulating attacks from malicious users and assessing their success [32]. This is accomplished by planning and reproducing an unauthorized user attack on the system, where the goal is to identify any vulnerabilities before a real attacker can make use of them. In this technique, vulnerability discovery is the same as exploiting them. Penetration testing can be divided into three categories based on the amount of information made available to the testers [32]:

- **Black-Box Penetration Testing**: assumes no prior knowledge of the infrastructure to be tested. This simulates an attack from someone unfamiliar with the system;
- **White-Box Penetration Testing**: provides the testers with complete knowledge of the infrastructure to be tested. This simulates an attack from someone with insider knowledge of the system (e.g an "inside job" or security leak);
- **Gray-Box Penetration Testing**: a variation that is somewhere between the previous two categories. This simulates an attack where there may have been a partial

disclosure of the system's inner workings, meaning testers will need to gather more information before employing this technique.

This technique can be further categorized based on how it's performed. Manual penetration testing is done without the aid of an automated tool, whereas automated penetration testing tools can be used to execute repetitive tasks and reduce testing time [8]. The first category can also be divided into systematic manual testing, where a predefined plan is followed, and exploratory manual testing, where this method is conducted with only the testers' instinct and experience. The diagram in Figure 2.3 provides an overview of penetration testing, illustrating the steps that are carried out by the testers while performing this technique.

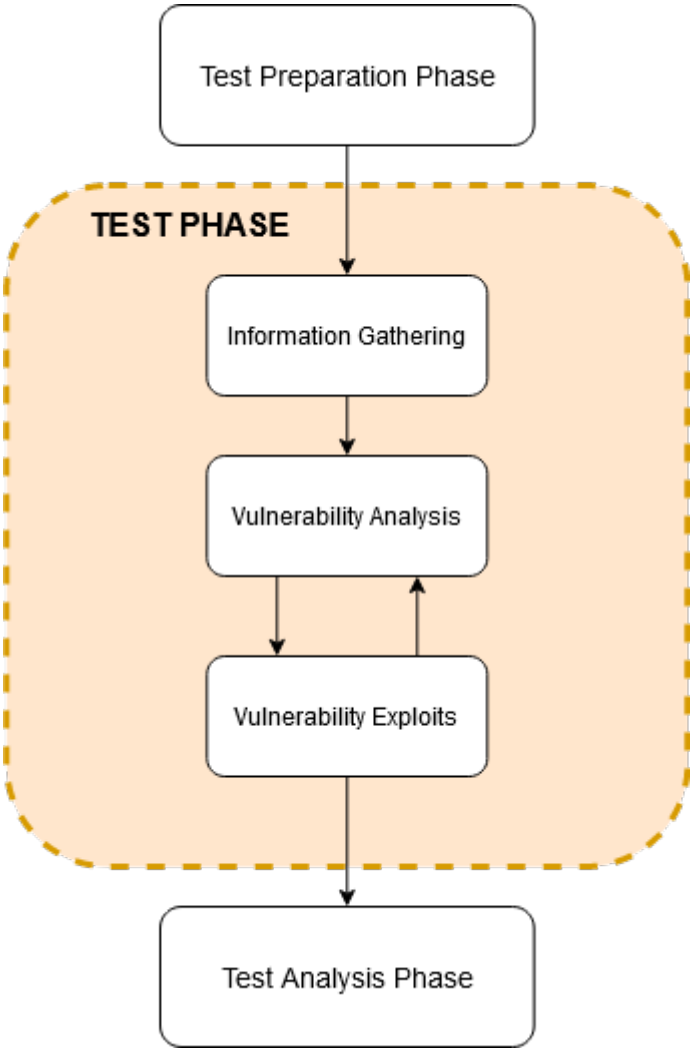


Figure 2.3: A diagram of the penetration testing methodology. Adapted from [9].

This process has several advantages including having no false positives, detecting design flaws, taking social engineering factors into account, and being able to expose vulnerabilities that can be hard to detect with other tools. Penetration testing can also focus on the workflow at all levels of an organization, allowing its members to know which parts of the system to prioritize and apply corrective measures [9]. These may prevent security breaches that would otherwise damage an organization's reputation which, as previously mentioned, could have significant customer loyalty and financial repercussions. On the other hand, it depends heavily on human interaction, whose results vary greatly depending on a person's

abilities, skills, and technical experience. Moreover, it may cause damage to the system in question while failing to detect vulnerabilities (false negatives). As new security threats and tools evolve, penetration testing must also be performed multiple times in the future to prevent more recent vulnerabilities from being exploited.

2.2.3 Fuzzing

Fuzzing is a technique that generates a semi-valid or random stream of data that is then sent to a running target application in order to test its resistance [32]. This semi-valid data is correct enough to pass input examinations, but still invalid enough to cause problems. The tool that generates this data, called fuzzer, has two proposed methods for doing so:

- **Data-Generation**: creates the data according to a given format or specification. This requires knowledge of the latter and needs a high amount of human involvement in the process. Additionally, this method traverses a high amount of program states (high code coverage) as it tries to create test cases for as many execution paths as possible [30];
- **Data-Mutation**: modifies the contents of valid input data, requiring little knowledge of any format or specification. When these are complex, data-mutation is favored over data-generation. However, data-mutation is not good enough for highly sensitive situations and heavily depends on the initial values, which means that different values may lead to very different code coverage rates and effects.

Detecting vulnerabilities with fuzzing can be advantageous as it has a high degree of automation while having no false positives. It also doesn't require access to the source code and can be easily scaled up to large applications. However, some drawbacks exist, including a large false negative rate, a low degree of generalization, and the long development cycle of fuzzing tools. Additionally, inputs may be processed with signature, encryption, or compression algorithms, requiring the fuzzer to also have the ability to perform these operations. Validating file formats and protocols can also be an obstacle to input generation.

2.3 Software Metrics

One other related construct used for detecting software faults and managing a project's quality over time would be software metrics. In order to keep track of the overall quality during software development, it becomes necessary to establish a process for quantifying and evaluating the project's performance. One way of accomplishing this is by measuring the intermediate product according to specific criteria so as to assess both a developer and their work [38]. This can take the form of values that characterize the source code itself (e.g. the number of **Lines of Code (LOC)** in a file), or whoever developed it (e.g. the number of defects introduced by a programmer per thousand LOCs) [21]. We will focus on the former, and will henceforth use "software metrics" to refer to source code metrics. Much like in SCA, their value can be determined without running the software.

As the programming languages used to create software were designed using different philosophies, called paradigms, there also exist distinct types of software metrics that accommodate each one [38]. The two most relevant paradigms in regard to software metrics

are procedural and object-oriented programming. In the former, a program's instructions are carried out sequentially and may be contained inside procedures, which can be called at any point during the execution (e.g. the C programming language). In the latter, a program uses the concept of objects, which contain their own data and procedures as defined by a class, to model a problem according to the relationship and interaction between other objects (e.g. the Java programming language). Some languages support both and are thus multi-paradigm, allowing the user to choose their own coding style (e.g. the C++ programming language). The three main software metrics properties from object-oriented literature, Complexity, Coupling, and Cohesion (CCC), are described below [14]:

- **Complexity**: where an object's complexity is described by the cardinality of its set of properties. For example, the number of inherited data in an object from its parents, or the number of different execution paths in a method. If either of these values were large enough, their respective object could be considered complex;
- **Coupling**: where two objects are considered coupled if at least one acts upon the other. For example, if one object accesses the data or invokes a method from the other;
- **Cohesion**: which describes the degree of similarity between two objects, i.e., the intersection between their sets of properties. For example, the similarity between two methods in an object can be seen as the number of common member variables that they access.

The same authors, Chidamber and Kemerer, used these properties to develop [14] and later improve [15] the following suite of object-oriented software metrics. These became some of the most used metrics in other studies [38], and are called the Chidamber and Kemerer (CK) metrics.

- **Weighted Methods Per Class (WMC)**: the sum of the complexities of every method in a class. This complexity is assumed to be one, and the sum equals the total number of methods in a class. This metric reflects an object's complexity since a higher value imply a greater amount of time developing and maintaining the class. It also means that any subclass will inherit a large number of methods, thus increasing their complexity;
- **Depth of Inheritance Tree (DIT)**: the number of parent classes that are inherited by a class. This metric reflects an object's complexity since the deeper its class is in the inheritance hierarchy, the greater the number of methods it can inherit;
- **Number of Children (NOC)**: the number of immediate subclasses that inherit a class. This metric reflects an object's complexity since the higher it is, the greater the number of children that will inherit the parent class' methods, requiring additional work to test them;
- **Coupling Between Objects (CBO)**: the number of other non-inherited classes that are coupled to a class. This metric reflects an object's coupling and how hard it would be to test it;
- **Response for a Class (RFC)**: the number of methods in a class that can potentially be executed, including inherited methods. This metric reflects an object's complexity since the greater the number of methods that may be invoked by other objects, the harder it is to test and debug;

- **Lack of Cohesion in Methods (LCOM)**: the number of disjoint sets formed by intersecting the sets of used member variables in every method in a class. This metric reflects an object's cohesion since fewer disjoint sets implies a larger degree of similarity between methods. If there are no used member variables in common, this value is zero.

For both procedural and object-oriented programming languages, one other equally famous complexity metric is **McCabe's Cyclomatic Complexity** [34]. This value counts the number of linearly independent execution paths of a procedure, and is computed by using the number of edges, nodes, and connected components from the program's control flow graph. By setting an upper limit to this metric, developers are forced to reduce the number of path combinations the execution may take, thus decreasing the amount of testing required to validate a program. Other variants of this metric also exist, including [16]:

- **Modified Cyclomatic Complexity**: identical to cyclomatic complexity, except that each switch statement (a construct where one branch is selected from several possible values) is assumed to have a complexity of one, instead of counting each individual case branch;
- **Strict Cyclomatic Complexity**: identical to cyclomatic complexity, except that, in each binary logical operator (AND and OR), every operand is also counted and is assumed to have a complexity of one;
- **Essential Cyclomatic Complexity**: measures the cyclomatic complexity by iteratively replacing well-structured control structures (e.g. IF-ELSE and WHILE constructs) with a single statement. This shows how much complexity remains once these well-structured constructs have been removed.

As was previously seen in the static and dynamic analysis sections, 2.1.1 and 2.2.1, it can be advantageous to keep track of how information flows through different components in a program. Henry and Kafura [26] proposed a complexity metric called the **Henry Kafura Size (HK)** which measures the interconnectivity of a procedure with its environment. This metric is computed by using the following three software metrics [26]:

- **Fan-In (FANIN)**: the number of inputs that a procedure takes, plus the number of global data structures that it reads information from;
- **Fan-Out (FANOUT)**: the number of outputs that a procedure emits, plus the number of global data structures that it writes information to;
- A metric that represents the length or complexity of the procedure such as LOC or cyclomatic complexity. This value may also be omitted from the metric's formula.

Where the final value of the HK metric is calculated using the formula:

$$Length \times (FANIN \times FANOUT)^2$$

This metric uses the product of FANIN and FANOUT to represent the total number of combinations from an input source to an output destination. This can be interpreted as

measuring the complexity of both the procedure and the relationship with its environment. Just as this value increases, so too does the number of implemented functionalities and the amount of information that flows through a procedure. A procedure of a large enough size could have too much influence on a system and may need to be broken down into smaller ones.

An example of a procedure and the values of its software metrics are presented in Listing 2.1 and in Table 2.1, respectively.

```
// This program solves the N-Queens Problem, where N queens must be
// placed on an NxN chessboard so that no two queens share the same
// row, column, or diagonal, i.e., they don't threaten each other.

int num_solutions = 0;
void solve_n_queens(int board_size, int column, int *column_history) {

  if(column == board_size) {
    printf("\nNo. %d\n-----\n", ++num_solutions);

    for(int i = 0; i < board_size; i++, putchar('\n'))
      for(int j = 0; j < board_size; j++)
        putchar(j == column_history[i] ? 'Q' : ((i + j) & 1) ? ' ' : '.');

    return;
  }

  #define UNDER_ATTACK(i, j) (column_history[j] == i || abs(column_history[
    j] - i) == column - j)

  for(int i = 0, j = 0; i < board_size; i++) {
    for(j = 0; j < column && !UNDER_ATTACK(i, j); j++);
    if(j < column) continue;

    column_history[column] = i;
    solve_n_queens(board_size, column + 1, column_history);
  }
}
```

Listing 2.1: A program written in the C programming language that finds and shows the solutions to the N-Queens problem. This code was adapted from the Rosetta Code website[1].

Metric	Value
LOC	15
Cyclomatic Complexity	9
Modified Cyclomatic Complexity	9
Strict Cyclomatic Complexity	11
Essential Cyclomatic Complexity	4
FANIN	5
FANOUT	5
HK	5625

Table 2.1: The values of several software metrics collected from the code in Listing 2.1. These were generated using a SAT called Understand[2].

Despite their simplicity and age, these types of metrics have been used to model software

[1] https://www.rosettacode.org/wiki/N-queens_problem#C
[2] https://www.scitools.com/

faults and defects [46, 49]. Software metrics that are easy to compute and understand are also favored over more academic metrics that are harder to collect in a real-world system [21]. Nuñez-Varela et al. [38] found that the ten previously mentioned metrics (LOC, the six CK metrics, FANIN, FANOUT, and cyclomatic complexity) were part of the top twelve most used software metrics from 226 different studies. In spite of their ubiquity, these metrics are defined at a micro-level, meaning they only apply to procedures or classes. These must be extended to the macro-level to draw conclusions about larger components such as entire source files [49]. One way to accomplish this is to aggregate them using operations like the sum, mean, median, maximum, and minimum functions. Although this may be done for any metric, if their distribution is skewed then the interpretation of the final result becomes unreliable. Additionally, software metrics may also differ in performance depending on what programming language is used [46].

2.4 Related Work: Software Vulnerability Detection Techniques

This section presents and summarizes other work related to detecting vulnerabilities with or without software metrics. In particular, we pay close attention to articles whose methodology tries to find vulnerabilities by using static analysis, machine learning algorithms, or both.

Harer et al. [25] presented a technique for detecting vulnerabilities in C/C++ code from Debian releases and GitHub repositories by generating machine learning models that classify functions as "good" or "buggy" by looking at both the source code and intermediate build files. The first approach uses natural language processing, which has no knowledge about the semantics of the languages, to analyze the source files, while the latter relies on the intermediate representation that is created when the programs are compiled. The source-based models used the following algorithms: the extremely random trees (extra-trees) classifier, a convolutional neural network called the TextCNN model, and a combination of these where the output of TextCNN is passed to the extra-trees classifier. Random forests were used for the build-based model.

These were evaluated using the Receiver Operating Characteristic (ROC) and the precision-recall curves, where the authors found that the best performance came from the model that combines both TextCNN and the extra-trees classifier. Both individual models had similar performances in the ROC metric. Additionally, build-based classifiers generally performed worse than source-based ones. The former also saw a very similar performance on the Debian and GitHub datasets. However, a model that combines both sets of source and build features was found to yield better results than either one, as the classifier had extra information to work with.

Algaith et al. [2] analyzed the performance of various SAT configurations when detecting SQL Injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities in 134 PHP plugins of the WordPress Content Management System. The goal was to run multiple SATs for the same source code, which would maximize and diversify the number of reported security alerts and thus result in a more complete vulnerability detection heuristic. Since separate SATs may be designed using different types of analysis, these tools' strengths and weaknesses would complement each other. In each studied configuration, the combination of N SATs could raise an alarm using one of the following three methods: 1-out-of-N, where the code is classified as vulnerable if at least one SAT classifies it as such; N-out-of-N, where the code is classified as vulnerable if all SATs classify it as such; or a simple majority vote.

This was done for the possible two, three, four, and five SAT combinations, where the tools used were RIPS, Pixy, phpSAFE, WAP, and WeVerca.

The metrics sensitivity, specificity, and ROC plots were used to measure this binary classifier's performance. Results showed that the RIPS and phpSAFE tools detected almost exclusively different vulnerabilities, meaning the usage of these two SATs would be advantageous as running both would cover a broader category of vulnerabilities than each one individually. Classifiers that used the 1-out-of-N detection method had better sensitivity when compared to individual SATs (0.96, 0.90, and 0.91 when using two, three, and five tools, respectively), whereas the N-out-of-N strategy presented better specificity (1.0 or very close for most tool configurations). For 1-out-of-N, this observation improved with the value of N, though this also resulted in worse specificity on average. Every XSS vulnerability was detected when all five SATs were considered in a 1-out-of-N approach. The majority voting classification showed a more balanced in-between version of these two methods. This last setup performed better than N-out-of-N in terms of sensitivity (0.342 for three tools), but worse for specificity (though with values still close to 1.0). Additionally, the ROC plots showed how most configurations have extremes of either high sensitivity (1-out-of-N) or specificity (N-out-of-N), as choosing one metric comes at the expense of the other. As the authors point out, choosing a configuration for the optimal classifier depends on an entity's use case. A 1-out-of-N approach is useful for detecting a wide array of vulnerabilities, though filtering true from false positives requires additional time and resources. Specifically, an organization may be interested in combining the tools RIPS, phpSAFE, and WAP, as they resulted in diverse vulnerability detection.

Chernis and Verma [12] researched how certain textual features performed when building classifiers that could label functions written in C as vulnerable or non-vulnerable. These functions were found by searching the National Vulnerability Database (NVD)[3] for open-source GitHub projects containing buffer overflow vulnerabilities and selecting one hundred for each label. Additionally, this information was complemented with functions from the top twenty most well-known Linux utilities. Two datasets were built, one mixed dataset which contained information from both sources, and one non-mixed which comprised solely of GitHub projects. The source files of these programs were parsed and two types of natural language processing features were extracted from their functions: trivial (function length, nesting depth, string entropy, etc) and non-trivial (n-grams and suffix trees). In this last category, the n-grams' statistics were fed to traditional machine learning algorithms, while the suffix trees were used to create a "stand-alone" classifier which assigned each function a vulnerable and non-vulnerable score and compared both values.

These other algorithms included naive bayes, k-nearest neighbors, k-means clustering, neural networks, Support Vector Machines (SVMs), decision trees, and random forests. For the trivial features, 5-fold cross-validation was used. This study found that word and character n-grams of lengths 1-4 and 1-5, respectively, resulted in an accuracy of 69%. On the other hand, the best suffix tree classifier only achieved 60% accuracy, even with ideal parameterization. When trivial features were considered, these reached a best of 75% accurate classifications. The SVM and k-means classifiers performed as well as naive bayes for character n-grams and character diversity, respectively. Moreover, by using the trivial features, the authors claimed that vulnerable functions were easier to be distinguished in the Linux tools than non-vulnerable ones were in the GitHub projects.

Siavvas et al. [45] investigated whether software metrics could discriminate between different vulnerability types and if there existed any interdependencies between them. The authors used the CKJM Extended tool to extract the CK and Quality Model for Object-

[3] `https://nvd.nist.gov/`

Oriented Design (QMOOD) [10] metrics from the top one hundred most popular Java libraries in the Maven repository. This last metrics suite looks at an object-oriented system's design (e.g. its class definitions and hierarchies) and measures its functionality, effectiveness, understandability, extendability, reusability, and flexibility. The rules generated by a SAT called PMD[4] were combined with Common Weakness Enumerations (CWEs) to group weaknesses into eight security-specific and five design-specific categories. Correlation analysis showed that the selected metrics were more closely related to security problems, and may be able to discriminate between vulnerabilities and quality-related issues. However, it was also observed that the correlation was weak or moderate within security-specific categories, meaning the selected software metrics may not be good indicators of specific vulnerabilities. This study also found that statistically significant correlations, and thus interdependencies, may exist between vulnerability categories.

Sultana [47] proposed a vulnerability prediction model that focused on traceable code patterns found in Java classes and methods, called micro and nano-level patterns, respectively. This classifier would be able to label classes and methods as vulnerable or neutral (where a previously reported vulnerability was patched) based on their patterns and software metrics. These features were collected from different versions of the Apache Tomcat and Apache CXF projects, and also from other open-source J2EE web applications in the Stanford Securibench dataset. A SAT called the Early Security Vulnerability Detector was used to identify vulnerable classes in this last dataset, while the tools JiraExtractor and Understand extracted nano-level patterns and software metrics, respectively, from the code. The author built the classifiers using the logistic regression and SVM algorithms with 10-fold cross-validation and analyzed their performance separately for the traceable patterns and software metrics features. The performance metrics used were false negative rate, precision, recall, and F-score. For patterns, the best results were 89.6% in recall and 83.1% in F-score (for SVMs using micro patterns), and 74.4% in precision (for logistic regression using nano-patterns). For software metrics, logistic regression using class metrics saw the best results for precision, recall, and F-score, with the values of 88.8%, 87.5%, and 87.8%, respectively.

This study found that some micro patterns had a greater statistically significant relation with neutral classes than with vulnerable ones. In a related example, a pattern where a class only has abstract methods and three or more static final fields of the same type had a more prominent presence in vulnerable code. By using the phi coefficient measure between pairs of patterns, several associated pairs were observed in both vulnerable and neutral code. Classifiers trained with the pattern features resulted in better vulnerability prediction with regards to recall and false negative rate, while this was only observed for the precision values when software metrics were considered instead.

2.5 Related Work: Application of Machine Learning Techniques to Software Vulnerability Detection

Until now, we have presented works that are generally related to detecting vulnerabilities using software metrics, SCA, or machine learning methods. This next section is of special importance as it lists papers whose contributions, be they a dataset or a framework, are directly applied to our methodology.

Alves et al. [5] explored the relationship between software metrics and the source code's vulnerabilities. The goal was to determine if these metrics can discriminate between vul-

[4] `https://pmd.github.io/`

nerable and neutral (non-vulnerable) code, and which ones were more prominent. Five large and widely used C/C++ projects were analyzed: the Linux Kernel, the software developed by Mozilla, the Xen Hypervisor, the Apache HTTP Server (httpd), and the GNU C Library (Glibc). These were chosen as they represent a broad class of software that are a common attack vector with a significant impact, allowing the authors to draw more general conclusions that would apply to similar applications. They are also open-source projects, meaning that software metrics can be extracted from their public repositories. The following methodology was used to build this dataset of software metrics and reported vulnerabilities from 2000 to 2016:

1. Retrieve information about the five project's vulnerabilities from various online platforms. These include CVE Details[5], which stores a wide array of Common Vulnerabilities and Exposures (CVEs), and project-specific security reports in websites like the Mozilla Foundation Security Advisories (MFSA)[6] and the Xen Security Advisories (XSA)[7];

2. Analyze the previous metadata to find and obtain the correct version of the vulnerable source code in each project's version control system (e.g. Git, Subversion, CVS);

3. Run the tool Understand to collect software metrics from the source code at both a macro-level (files) and micro-level (functions and classes), before and after the vulnerability was fixed (patched);

4. Store the previously obtained metrics and vulnerability information in a database that was specifically designed to hold and relate this type of data;

5. Perform statistical analysis in this newly built dataset in order to find any relationships between the collected metrics and vulnerabilities.

This study found that some software metrics were highly correlated, meaning they contain redundant information and should be reduced to a single one. Not doing so could otherwise be an issue when building predictive models in the future. By applying statistical tests with two hypotheses, the authors showed that vulnerable code has different properties than neutral code. Thus, software metrics are able to represent vulnerable functions. The study also looked at the correlation between software metrics and individual functions, and obtained no high correlations (positive or negative). This means that there is no single metric that is able to determine if a function will have more vulnerabilities. Future predictive models must then combine multiple metrics in order to produce reliable results. Finally, functions with vulnerabilities were shown to have a tendency to repeat since the probability of the same function having two vulnerabilities was fifty-five times higher than the same probability if these vulnerabilities were not correlated. As such, vulnerable functions are more prone to obtain more vulnerabilities.

The same authors later [6] used this dataset to evaluate various machine learning techniques to detect vulnerabilities, so as to find the best models to predict them. The results varied, with the best algorithms being decision trees (when prioritizing by high precision) and logistic regression (when prioritizing by high recall). The accuracy metric did not prove to be very useful for portraying the effectiveness of these kinds of techniques. Due to bias limitations of precision and recall, the informedness and markedness unbiased metrics

[5] `https://www.cvedetails.com/`
[6] `https://www.mozilla.org/en-US/security/advisories/`
[7] `https://xenbits.xen.org/xsa/`

were analyzed. Tree-based approaches, such as random forests, performed better than the others in terms of informedness.

Medeiros et al. [35] studied whether software metrics could be used to distinguish vulnerable and non-vulnerable code at a file and function level, and showed how a near best subset of these metrics could be found. This was done by performing a heuristic search, which combined genetic algorithms with random forest classifiers, using the previously mentioned dataset [5] of software metrics and vulnerabilities in five major C/C++ projects. In this search, a population of software metrics (features) was iterated a number of times, and metrics were selected and removed in order to maintain or improve the classifier's overall performance. The accuracy and Cohen's Kappa metrics are used for this purpose, where this last one represents how much better or worse the classifier is versus random chance. The information from individual and collective projects was used, allowing for the creation of models that are accurate for specific software.

By using the Pearson and Spearman correlation coefficients, this study found that the sum of the essential cyclomatic complexity and CBO metrics had a very strong positive relationship with the number of vulnerabilities at a project level. Others, like the number of functions, FANOUT, HK, and the sum of the modified cyclomatic complexity also had a strong monotonic relation with this value. Similarly, the LCOM metric presented a strong positive linear relationship with vulnerable code. At a file level, the genetic algorithm converged to the best accuracy of 97.66% seven out of ten times. Here, complexity metrics were in the majority of solutions. Conversely, at the function level, volume metrics such as LOC and the number of commented lines were in the majority. For this code unit, the Kappa values were lower than at a file level, where the best accuracy and Kappa results were 97.30% and 37.03%, respectively.

Campos et al. [11] addressed the problem of having a cumbersome experimental machine learning workflow by presenting their own generalizable framework called Propheticus[8]. Because researchers may be forced to spend time implementing potentially error-prone validation code, this tool allows them to more easily try out different combinations of machine learning algorithms and preprocessing techniques, and choose the best configuration according to a given performance metric. Although showcasing this framework was the authors' main purpose, this paper uses part of the dataset developed in [5] as one of two case studies. Moreover, as is later discussed in Chapter 3, our work also makes use of this tool.

The authors considered only software metrics in files from the Mozilla project and chose a combination of techniques that would improve the previously mentioned informedness performance metric. The Propheticus tool was executed using the following machine learning algorithms: SVMs, gradient boosting, bagging, decision trees, AdaBoost, extra trees, neural networks, random forests, and k-nearest neighbors. Results yielded a correct prediction of 82% and 75% for random forests and bagging when predicting vulnerable and non-vulnerable samples, respectively, and using random undersampling as the data balancing approach. As the precision was low, a combination of the Synthetic Minority Oversampling Technique (SMOTE) data balancing technique with feature selection by correlation changed the previous percentages to 88% and 69%, respectively, with fewer false positives for the random forests algorithm. By experimenting with different hyperparameters, the best prediction results ended with 81% and 77%, respectively, when balancing the dataset with Instance Hardness Threshold and using random forests with feature selection by variance and correlation.

[8] `http://www.joaorcampos.com/ISSRE-2019/`

Pereira and Vieira [40] explored the results of two SATs, Cppcheck and Flawfinder, in a large C/C++ codebase developed by Mozilla by measuring their performance and applicability across different vulnerability categories. The authors used the previously mentioned dataset from [5]. In this study, numeric identifiers known as CWEs were assigned different categories based on the best security practices defined by the Open Web Application Security Project (OWASP), allowing the authors to better group and understand multiple vulnerability types.

The authors found that the two SATs were not able to detect about a quarter of the vulnerabilities, meaning that relying solely on SCA is not enough. As is expected of SATs, they also reported a high number of false positives, with Flawfinder reaching 94% of all generated alerts. However, it was noted that some false positives may actually be real undetected vulnerabilities that were not reported in Mozilla's bug tracker or on the CVE Details website. The number of these alerts increased over time, which was expected since Mozilla's codebase implemented more functionalities over time.

The performance metrics precision, recall, and F-score were used. Cppcheck had better results than Flawfinder when taking into account all vulnerabilities. When looking at specific categories, these values fluctuated but were still low, meaning none of the SATs could effectively detect vulnerabilities of different types. As such, this study concluded that SATs were not effective for detecting different categories of vulnerabilities in large-scale projects.

Pereira et al. [41] combined software metrics with security alerts generated by SATs to create machine learning models capable of distinguishing between vulnerable and non-vulnerable files, or assigning them a specific vulnerability category. Once again, the dataset developed by [5] was used, though only files from the Mozilla project were considered. The aim of this study was to check whether using both metrics and alerts could improve the trained models, to see if categorizing vulnerabilities would improve the machine learning algorithms, and to explore which code properties helped or hindered the classification process. Three datasets were defined, using just software metrics as their features, just security alerts, or both. This same number of experiments were conducted using the following labels: binary (vulnerable or non-vulnerable), binary within each vulnerability category, and multiclass (vulnerable in a specific category, non-vulnerable, or vulnerable without a category).

The aforementioned Propheticus tool was used to train the machine learning models, where the decision tree, random forest, extreme gradient boosting, and bagging algorithms were selected. For the binary class, the best precision was 0.9404 when only software metrics were considered, while the highest recall was 0.9019 for the dataset containing both alerts and metrics. In the binary per category class, the best precision was obtained when considering only software metrics for the memory management and permission categories (0.8447 and 0.8889, respectively). For the remaining input validation class, this value was at its highest (1.0) when using both metrics and alerts. In a similar manner, the best recall values for all categories (0.8826, 0.8956, and 0.8852, respectively) were observed when the metrics-only dataset was used. The bagging algorithm yielded all the previously mentioned results, with the exception of the recall value in memory management, where extreme gradient boosting was employed. Finally, for the multiclass classifiers, the highest precision and recall (not counting the non-vulnerable class which comprised most of the dataset) were 0.8347 and 0.6202 for the system configuration and no category classes, respectively.

The authors concluded that, not only could security alerts not be used to predict vulnerable Mozilla files, but also that combining them with software metrics did not yield

any significantly different results. Additionally, the trained machine learning models did not perform any better when using categories rather than just a binary label. This was explained by there existing more samples, and thus more information to train the models, with a binary label as opposed to cases with multiple possible categories. By exploring the files themselves, the authors also noted that 20.37% of non-vulnerable samples shared at least eleven metric values with their vulnerable counterparts, potentially impacting the observed results.

Although SCA alone may not be enough to robustly discover vulnerabilities [40], the usage of SATs has assisted their detection both directly and indirectly with varying degrees of success [2, 40, 41, 45, 47]. Likewise, though no single software metric is able to determine if a given function is more vulnerable [5], these static properties have been used to identify vulnerable code or distinguish between different types of vulnerabilities [5, 6, 35, 45, 47]. While the software metrics and vulnerabilities dataset developed by Alves et al. [5] has been applied to some of these studies [6, 11, 35, 40, 41], the information it contains as since been out of date. Just as new vulnerabilities are disclosed and made available in online platforms, so too does the need to gather and apply this information in a timely fashion increase. Therefore, we describe a process in Chapter 3 that automates the collection of both software metrics and security alerts reported by different SATs from open-source projects. The dataset presented by these authors is then updated and extended, allowing newly gathered vulnerabilities to be used to build models capable of identifying vulnerable and non-vulnerable code.

Chapter 3

Building the Vulnerable Code Unit Datasets

This chapter presents each step that was carried out in order to implement a mechanism capable of aiding vulnerability detection by automating the collection of resources available in online platforms and in an application's source code. This includes retrieving vulnerability metadata from websites, interfacing with a C/C++ project's version control system, generating software metrics and security alerts through Static Code Analysis (SCA), and aggregating the collected information into a dataset capable of being processed by machine learning algorithms.

Although we saw different kinds of Static Analysis Tools (SATs) in Sections 2.4 and 2.5, the term SAT is henceforth used to refer exclusively to tools that report possible vulnerabilities in source code, otherwise known as security alerts. Tools that generate software metrics from code units (files, functions, classes) using SCA are employed in our work, but they are referred to by name rather than using the term SAT.

3.1 Overview

The series of steps implemented during this work's development are similar to what was done by Alves et al. [5] to develop their own dataset. This dataset contains information from five large and widely used C/C++ projects: the Linux Kernel, the software developed by Mozilla, the Xen Hypervisor, the Apache HTTP Server (httpd), and the GNU C Library (Glibc). However, as its vulnerability metadata ranges from 2000 to 2016, it is now out of date. As modern applications require continuous security testing throughout their life cycle [22], a fast and automated mechanism for detecting software vulnerabilities is necessary. To ensure that any current and future data is added to this knowledge base, our process was developed with automation in mind.

Additionally, this dataset's design is extended to allow the introduction of new entities, namely security alerts, SAT properties, and other security related concepts like Common Weakness Enumerations (CWEs). This is akin to what was seen in Pereira and Vieira [40] and Pereira et al. [41]. By validating the resulting dataset with machine learning techniques, not only will we be able to assess our own work, but we will also be creating classifiers capable of quickly labeling a code unit as vulnerable or non-vulnerable. In a real-world scenario, having to review every single file, function, or class can be cumbersome, especially in large codebases. By narrowing down this number to a select few potentially

vulnerable code units, a great deal of development time and resources can be saved.

The five C/C++ projects used by both the previously mentioned authors and our implementation are summarized in Table 3.1. These projects were chosen by the original authors as they represent a broader class of software that are usually attack vectors exploited by malicious users (such as operating systems, web servers, web browsers, standard libraries, etc) meaning their data should apply to similar software. Although this table makes a reference to the SVN version control system (for Apache only), in practice we only interfaced with the Git mirror of each project. Unless noted otherwise, any future mention of a commit or repository refers to the Git version control system.

Project	**Language**	**Version Control**	**Lines of Code**	**Website**
Mozilla	C++	Git[1]	22.7 million	Mozilla.org
Linux Kernel	C	Git[2]	20 million	Kernel.org
Xen Hypervisor	C	Git[3]	0.6 million	XenProject.org
Apache HTTP Server	C	SVN[4]and Git[5]	1.5 million	Apache.org
GNU C Library (Glibc)	C	Git[6]	1.2 million	Gnu.org

Table 3.1: A summary of the five large C/C++ projects used in our work. The total number of lines of code was taken from the Open Hub website[7]on January 2021.

In this chapter, we will show how datasets composed of security alerts and software metrics obtained via SCA techniques can be built. This data structure is developed based on information that is first collected from online sources and later analyzed by third-party tools. A very brief overview of these steps, all of which are explained in depth in the subsequent sections, is described below:

i. Retrieve reported vulnerabilities present in open-source projects by querying online vulnerability databases and parsing their responses;

ii. Use this metadata to find and retrieve the correct vulnerable source files from each project's version control system (e.g. Git, Subversion, CVS);

iii. Run specific tools to collect security alerts and software metrics, before and after the vulnerability was fixed;

iv. Store these alerts, metrics, and any other information in a database designed specifically to house and relate these entities;

v. Aggregate the previously collected data into datasets and validate them by developing machine learning models capable of labeling a code unit (a file, function, or class) as vulnerable, non-vulnerable, or belonging to a specific vulnerability category.

A graphical representation of this process can be seen in Figure 3.1. We will be referring to this image throughout each section for the reader's convenience. This chapter will not go into specific implementation details. For a complete description of every script developed for this work, refer to the documentation in Appendix 6.

[1] `https://github.com/mozilla/gecko-dev`
[2] `https://github.com/torvalds/linux`
[3] `https://xenbits.xen.org/gitweb/?p=xen.git;a=summary`
[4] `https://svn.apache.org/repos/asf/httpd/httpd/trunk/`
[5] `https://github.com/apache/httpd`
[6] `https://sourceware.org/git/glibc.git`
[7] `https://www.openhub.net/`

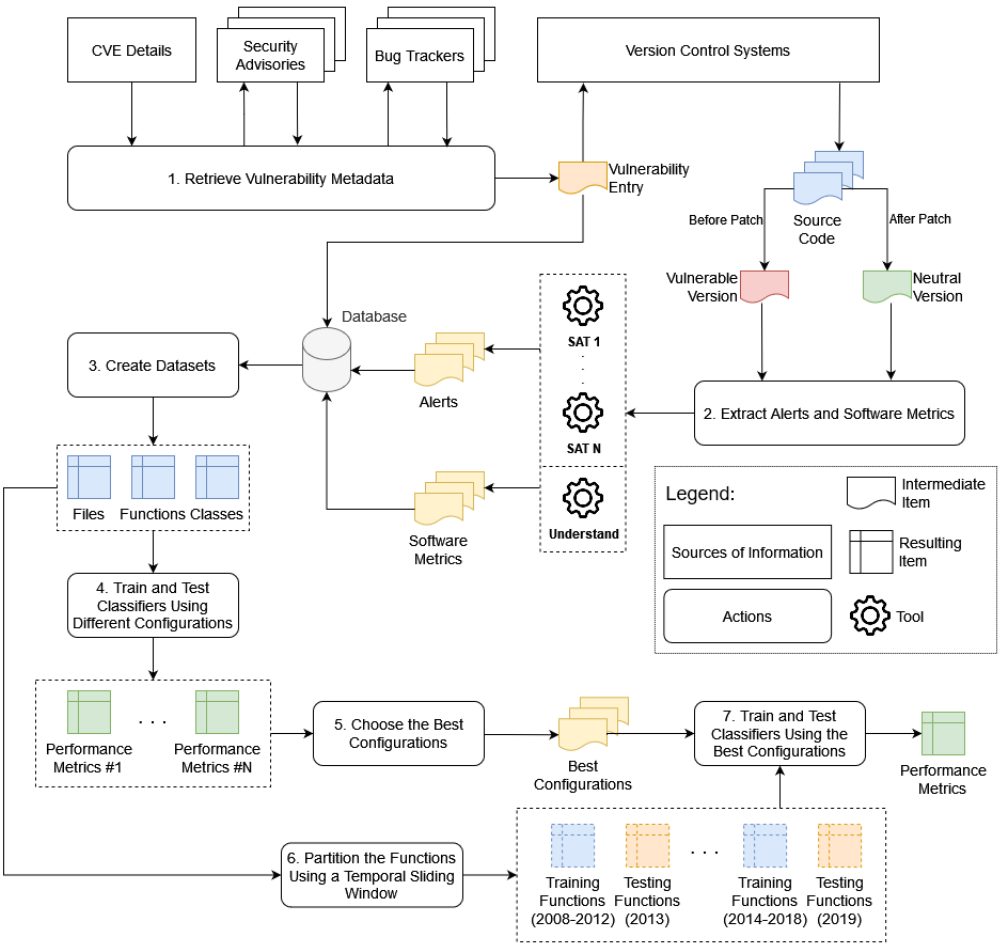


Figure 3.1: A diagram showing the full dataset creation and validation pipeline. This process begins with retrieving the reported vulnerabilities' metadata, and ends with validating the generated dataset.

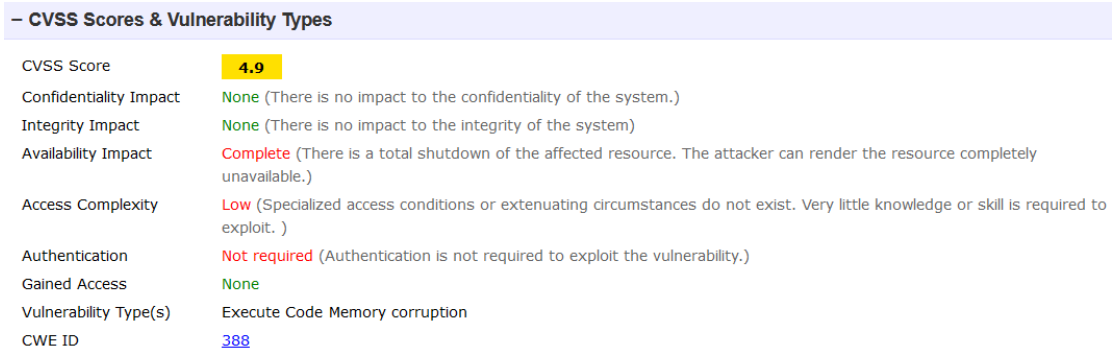
3.2 Retrieving Reported Vulnerabilities from Online Platforms

In this stage, we start by querying the CVE Details website[8] to obtain a list of reported vulnerabilities for the five projects. This is done by first requesting a table of Common Vulnerabilities and Exposure (CVE) entries from the vulnerability list endpoint[9] using each project's parameters, where each row contains a hyperlink to the vulnerability itself. Since each of these tables only lists a set number of CVEs, we must traverse to the next page and repeat this data scraping process.

For each vulnerability, its page is analyzed and any relevant fields in the metadata are saved. As can be seen in Figure 3.2, vulnerabilities contain, among other details: a unique identifier called the CVE; a value known as the Common Vulnerability Scoring System (CVSS) score which represents how severe it is given its properties and environment [52]; how much it impacts the confidentiality, integrity, and availability of a system; how hard it is to exploit; whether or not authentication is required to exploit it; zero or more vulnerability types; and an optional numerical identifier known as a CWE which maps a

[8] https://www.cvedetails.com/
[9] https://www.cvedetails.com/vulnerability-list.php

vulnerability to a specific category.



– CVSS Scores & Vulnerability Types

CVSS Score	4.9
Confidentiality Impact	None (There is no impact to the confidentiality of the system.)
Integrity Impact	None (There is no impact to the integrity of the system)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Execute Code Memory corruption
CWE ID	388

Figure 3.2: A screenshot of the score table for CVE-2018-1000199 in the CVE Details website[10], last accessed on January 2021.

– References For CVE-2019-15215

https://usn.ubuntu.com/4118-1/
UBUNTU USN-4118-1

https://syzkaller.appspot.com/bug?id=b68d3c254cf294f8a802582094fa3251d6de5247

https://usn.ubuntu.com/4115-1/
UBUNTU USN-4115-1

https://security.netapp.com/advisory/ntap-20190905-0002/ CONFIRM

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=eff73de2b1600ad8230692f00bc0ab49b166512a

http://www.openwall.com/lists/oss-security/2019/08/20/2
MLIST [oss-security] 20190820 Linux kernel: multiple vulnerabilities in the USB subsystem x2

https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.2.6

Figure 3.3: A screenshot of the references table for CVE-2019-15215 in the CVE Details website[11], last accessed on January 2021.

Additionally, each CVE page lists several relevant external references, including the software's changelog, discussion boards, the project's bug tracker, and the version control. An example of this can be seen in Figure 3.3, where this section presents the references of a vulnerability in the Linux Kernel that was published in 2019 and whose identifier is CVE-2019-15215. In this image, notice how the CVE page links to a Git repository and how its Uniform Resource Locator (URL) contains a unique identifier called a commit hash. This value allows us to locate any files affected by this vulnerability in a given version (i.e. commit) of the software. As such, they are necessary in order to retrieve the files themselves and apply SCA. For this example, CVE-2019-15215 affects a single file in the version represented by the hash "eff73de2b1600ad8230692f00bc0ab49b166512a" from the Linux Kernel's Git repository. For other projects, if this identifier is not readily available then it is necessary to consult other references. Two systems that can be used to extract additional data are bug trackers and security advisories.

Bug trackers help developers keep track of bugs in their software, allowing users to submit and discuss reports. As these bug reports often point to the previously mentioned commit hashes, we can make use of them if they are linked in a CVE's external references. In particular, we noticed how, in some projects, the bug tracker identifier is present in the message of the commit that addresses the respective bug. Because of this, we are able to find any commit hashes related to a vulnerability given its bug tracker identifier. Table 3.2

[10] `https://www.cvedetails.com/cve/CVE-2018-1000199/`
[11] `https://www.cvedetails.com/cve/CVE-2019-15215/`

shows the projects where this process could be applied, as well as the regular expression patterns that were used to retrieve a commit's hash from its message.

A security advisory is a system that publishes information about vulnerabilities regarding a specific developer or product. From these platforms, we can extract more specific information that may be absent from CVE Details, including what particular applications and versions are vulnerable, a brief description of the vulnerability, any known workarounds to avoid them, other closely associated CVEs, and additional references such as bug reports. Much like with bug trackers, the advisory identifier associated with each entry can be used to find a commit's hash given its message. As can be seen in Table 3.2, this process can only be applied to the Xen Security Advisories (XSA).

Examples of the commit messages mentioned in the previous two use cases can be seen in Figures 3.4 and 3.5. Notice how the text in both screenshots matches the pattern of the respective project in Table 3.2. The Linux Kernel project does not have a bug tracker or security advisory listed since its commit hashes are all retrieved using the references section in the CVE Details website.

Project	Bug Tracker	Security Advisory	Commit Message Pattern
Mozilla	Mozilla.org	Mozilla.org	`^Bug \b<BUG_ID>\b`
Linux	-	-	-
Xen	-	Xen.org	`This is.*\b(<CVE>|<ADV_ID>)\b`
Apache	Apache.org	Apache.org	`SECURITY:.*\b<CVE>\b`
Glibc	Sourceware.org	-	`((BZ|Bug).*\b<BUG_ID>\b)|(\bBZ<BUG_ID>\b)`

Table 3.2: A summary of the bug tracker and security advisory websites considered for the C/C++ projects, as well as the regular expressions used to retrieve a commit's hash via its message. The tokens `<CVE>`, `<BUG_ID>`, and `<ADV_ID>` refer to the CVE, bug tracker, and security advisory identifiers, respectively.

Figure 3.4: A screenshot of the commit message associated with the advisory identified by XSA-87 from the Xen project, in the GitHub website[12], last accessed on October 2021.

Before moving to the next step, we finish the vulnerability collection process by removing any commit hashes that are not valid (because they may belong to a different project), as well as any commits outside of the repository's main branch. Part of the information collected via this mechanism can be seen in Figure 3.6, where a file containing the previously mentioned metadata is shown. The process described in this section is represented by the "Retrieve Vulnerability Metadata" action in the top left corner of the diagram in Figure 3.1.

[12] https://github.com/xen-project/xen/commit/9c7e789a1b60b6114e0b1ef16dff95f03f532fb5
[13] https://github.com/bminor/glibc/commit/7c1f4834d398163d1ac8101e35e9c36fc3176e6e

Figure 3.5: A screenshot of the commit message associated with the bug identified by 13656 from the Glibc project, in the GitHub website[13], last accessed on October 2021.

CVE	CVSS Score	Integrity Impact	Access Complexity	Authentication	Vulnerability Types	CWE	Affected Product Versions	Bugzilla IDs	Advisory IDs	Git Commit Hashes
CVE-2018-5145	7.5	Partial	Low	Not required	["Overflow", "Memory corr...	119	{"Firefox Esr": ["10.0", "10.0...	["1261175,...	["MFSA-2018-07", "MFSA-...	
CVE-2018-5144	7.5	Partial	Low	Not required	["Overflow"]	190	{"Firefox Esr": ["10.0", "10.0...	["1440926"]	["MFSA-2018-07", "MFSA-...	
CVE-2018-5143	4.3	Partial	Medium	Not required	["Cross Site Scripting"]	79	{"Firefox": ["0.1", "0.2", "0.3...	["1422643"]	["MFSA-2018-06"]	["11045868647d8...
CVE-2018-5142	5.0	Partial	Low	Not required			{"Firefox": ["0.1", "0.2", "0.3...	["1366357"]	["MFSA-2018-06"]	["15353262e8f3d7...
CVE-2018-5141	6.4	None	Low	Not required	["Denial Of Service"]	20	{"Firefox": ["0.1", "0.2", "0.3...	["1429093"]	["MFSA-2018-06"]	["49a4cb64e9a101...
CVE-2018-5140	5.0	None	Low	Not required	["Obtain Information"]	200	{"Firefox": ["0.1", "0.2", "0.3...	["1424261"]	["MFSA-2018-06"]	["3c59d52401660e...
CVE-2018-5138	5.0	Partial	Low	Not required		20	{"Firefox": ["0.1", "0.2", "0.3...	["1432624"]	["MFSA-2018-06"]	["1c29a391dfa4ac...
CVE-2018-5137	5.0	None	Low	Not required	["Obtain Information"]	200	{"Firefox": ["0.1", "0.2", "0.3...	["1432870"]	["MFSA-2018-06"]	["714c763d13076f...
CVE-2018-5136	5.0	None	Low	Not required	["Bypass a restriction or si...	20	{"Firefox": ["0.1", "0.2", "0.3...	["1419166"]	["MFSA-2018-06"]	["2b3039c02109b...
CVE-2018-5135	5.0	Partial	Low	Not required	["Bypass a restriction or si...	862	{"Firefox": ["0.1", "0.2", "0.3...	["1431371"]	["MFSA-2018-06"]	["8026fd51f12cfb9...
CVE-2018-5134	5.0	None	Low	Not required	["Bypass a restriction or si...	200	{"Firefox": ["0.1", "0.2", "0.3...	["1429379"]	["MFSA-2018-06"]	["1d8cd72e69474...
CVE-2018-5133	4.3	None	Medium	Not required	["Obtain Information"]	200	{"Firefox": ["0.1", "0.2", "0.3...	["1430974",...	["MFSA-2018-06"]	["4b11b9c788f433...
CVE-2018-5132	4.3	None	Medium	Not required	["Obtain Information"]	200	{"Firefox": ["0.1", "0.2", "0.3...	["1408194"]	["MFSA-2018-06"]	["0fb829b692d4cf...
CVE-2018-5131	4.3	None	Medium	Not required	["Obtain Information"]	200	{"Firefox": ["0.1", "0.2", "0.3...	["1440775"]	["MFSA-2018-07", "MFSA-...	["6a92d27239f7bd...
CVE-2018-5130	6.8	Partial	Medium	Not required		20	{"Firefox": ["0.1", "0.2", "0.3...	["1433005"]	["MFSA-2018-07", "MFSA-...	["ef3c58e0e688d8...
CVE-2018-5129	5.0	Partial	Low	Not required	["Memory corruption"]	787	{"Firefox": ["0.1", "0.2", "0.3...	["1428947"]	["MFSA-2018-09", "MFSA-...	["35b53a22df6d44...
CVE-2018-5128	7.5	Partial	Low	Not required		416	{"Firefox": ["0.1", "0.2", "0.3...	["1431336"]	["MFSA-2018-06"]	["8a6cab48a92c04...
CVE-2018-5127	6.8	Partial	Medium	Not required	["Overflow"]	119	{"Firefox": ["0.1", "0.2", "0.3...	["1430557"]	["MFSA-2018-09", "MFSA-...	
CVE-2018-5126	7.5	Partial	Low	Not required	["Overflow", "Memory corr...	119	{"Firefox": ["0.1", "0.2", "0.3...	["1433671,...	["MFSA-2018-06"]	
CVE-2018-5125	6.8	Partial	Medium	Not required	["Overflow", "Memory corr...	119	{"Firefox": ["0.1", "0.2", "0.3...	["1416529,...	["MFSA-2018-09", "MFSA-...	
CVE-2018-5124	4.3	Partial	Medium	Not required	["Execute Code", "Cross Sit...	79	{"Firefox": ["0.1", "0.2", "0.3...		["MFSA-2018-05"]	

Figure 3.6: An example of the Comma-Separated Values (CSV) file generated after collecting the vulnerability metadata from CVE Details and other referenced websites for the Mozilla project.

3.3 Retrieving Vulnerable Source Files from Version Control

Following the previous stage, we use the Git commit hashes obtained when collecting vulnerability metadata to find the location of any affected files in each project's repository. This is done by interfacing with its version control system and requesting the files for a specific version. Although this method would still apply to different version control systems, our work only considered the Git repositories listed in Table 3.1. Git's `rev-list`[14] command is used to list every file path affected by a vulnerability, as well as each commit's tag name, author date, and a set of line number ranges that show where each file was modified. This file list is then shortened by only considering the following C/C++ source file extensions: `.c`, `.cpp`, `.cc`, `.cxx`, `.c++`, `.cp`, `.h`, `.hpp`, `.hh`, `.hxx`.

After finding both the vulnerable software version and location of the affected files, these must now be retrieved from their repository in order to conduct any kind of SCA. One may accomplish this step by first cloning the repository, which recreates its original directory structure and copies every file it contains. Following that, the commit hash identifiers found in Section 3.2 are used to restore the repository to specific versions (commits) using the Git `checkout`[15] command. Finally, we use the same approach as Alves et al. [5] and Pereira and Vieira [40], and obtain one version of the repository where the vulnerability was corrected (patched) and another immediately before this patch.

[14] `https://git-scm.com/docs/git-rev-list`
[15] `https://git-scm.com/docs/git-checkout`

By following both of these steps, we now have two versions of the affected files, one vulnerable (before it was patched) and one non-vulnerable (after it was patched, i.e., neutral). An example of a file in a Git repository, before and after it was patched, is shown in Figure 3.7. Notice how we can also see where the file was modified in the vulnerable and neutral commits (the lines 905 and 913-914, respectively).

Figure 3.7: A screenshot of the differences in the file affected by CVE-2019-15215, before and after it was patched, in the GitHub website[16], last accessed on January 2021.

Given that we have retrieved the files themselves at this point, we take this opportunity to also enrich the data we have been collecting with a list of functions and classes in each source file. This is done by running the Clang compiler[17] which will parse the C/C++ source files and generate an Abstract Syntax Tree (AST) comprised of nodes that represent code constructs. By traversing this structure while using the line number ranges obtained above, we can create a list of functions and classes that show whether each of these code units is vulnerable or not.

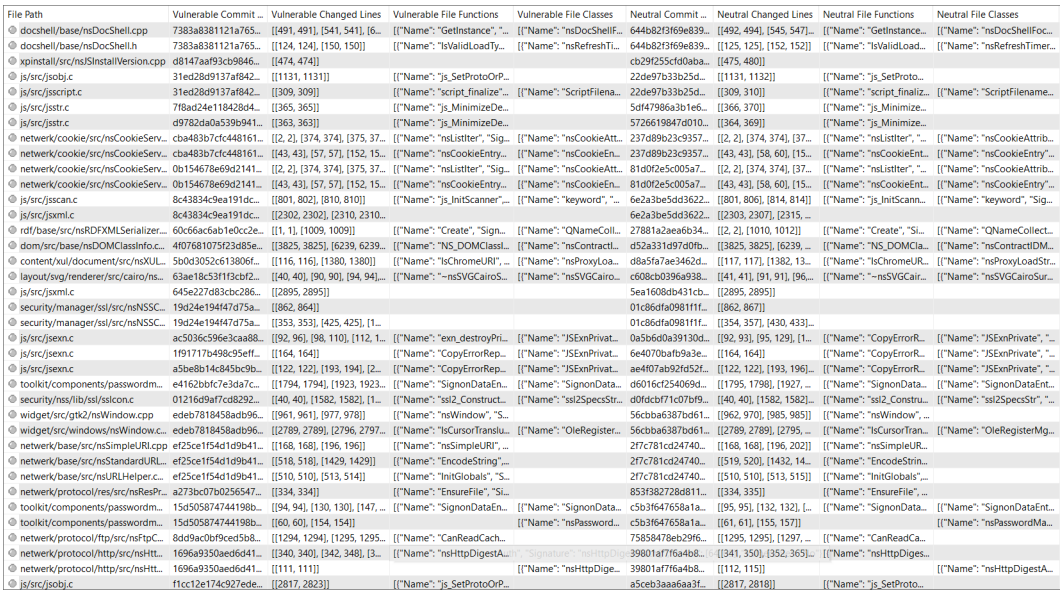
Any file present in a neutral commit or a commit not affected by a vulnerability is assumed to be neutral. Likewise, its code units would also be non-vulnerable. For a vulnerable commit, each affected file is assumed to be vulnerable, but the same might not be true of its code units. For example, a vulnerable file that has five functions may have only been patched in one of them. Because of this, we take extra care in checking whether the commit's line number ranges overlap with any code units inside vulnerable files.

To ease any future SCA operation, we follow one final step and generate what we call a file timeline. This structure lists every single file path of every commit (affected by a vulnerability or not) according to the main branch's topological order. While the information in this timeline is essentially a copy of what was already mentioned in this section, the process of going through every commit in order helps in removing any erroneous file paths. Consider the case of two successive pairs of vulnerable and neutral commits where the same vulnerability was patched twice. Here, the first neutral commit is the same as the second vulnerable commit. If no adjustments are done, then we may be listing files that turned out to be vulnerable at a later date as being neutral.

Examples of the affected file metadata as well as the file timeline can be seen in Figures 3.8 and 3.9, respectively. The process of retrieving a vulnerable and neutral version of each

[1] `https://github.com/torvalds/linux/commit/eff73de2b1600ad8230692f00bc0ab49b166512a`
[17] `https://clang.llvm.org/`

file affected by a vulnerability is represented by the "Version Control Systems" source of information on the top right corner of the diagram in Figure 3.1.



File Path	Vulnerable Commit ...	Vulnerable Changed Lines	Vulnerable File Functions	Vulnerable File Classes	Neutral Commit ...	Neutral Changed Lines	Neutral File Functions	Neutral File Classes
docshell/base/nsDocShell.cpp	7383a8381121a765...	[[491, 491], [541, 541], [6...	[{"Name": "GetInstance", "...	[{"Name": "nsDocShellF...	644b82f3f69e839...	[[492, 494], [545, 547]...	[{"Name": "GetInstance...	[{"Name": "nsDocShellFoc...
docshell/base/nsDocShell.h	7383a8381121a765...	[[124, 124], [150, 150]]	[{"Name": "IsValidLoadTy...	[{"Name": "nsRefreshTi...	644b82f3f69e839...	[[125, 125], [152, 152]]	[{"Name": "IsValidLoad...	[{"Name": "nsRefreshTimer...
xpinstall/src/nsJSInstallVersion.cpp	d8147aaf93cb9846...	[[474, 474]]			cb29f255cfd0aba...	[[475, 480]]		
js/src/jsobj.c	31ed28d9137af842...	[[1131, 1131]]	[{"Name": "js_SetProtoOrP...		22de97b33b25d...	[[1131, 1132]]	[{"Name": "js_SetProto...	
js/src/jsscript.c	31ed28d9137af842...	[[309, 309]]	[{"Name": "script_finalize"...	[{"Name": "ScriptFilena...	22de97b33b25d...	[[309, 310]]	[{"Name": "script_finaliz...	[{"Name": "ScriptFilename...
js/src/jsstr.c	7f8ad24e118428d4...	[[365, 365]]	[{"Name": "js_MinimizeDe...		5df47986a3b1e6...	[[366, 370]]	[{"Name": "js_Minimize...	
js/src/jsstr.c	d9782da0a539b941...	[[363, 363]]	[{"Name": "js_MinimizeDe...		5726619847d010...	[[364, 369]]	[{"Name": "js_Minimize...	
netwerk/cookie/src/nsCookieServ...	cba483b7cfc448161...	[[2, 2], [374, 374], [375, 37...	[{"Name": "nsListIter", "Sig...	[{"Name": "nsCookieAtt...	237d89b23c9357...	[[2, 2], [374, 374], [37...	[{"Name": "nsListIter", "...	[{"Name": "nsCookieAttrib...
netwerk/cookie/src/nsCookieServ...	cba483b7cfc448161...	[[43, 43], [57, 57], [152, 15...	[{"Name": "nsCookieEntry...	[{"Name": "nsCookieEn...	237d89b23c9357...	[[43, 43], [58, 60], [15...	[{"Name": "nsCookieEnt...	[{"Name": "nsCookieEntry"...
netwerk/cookie/src/nsCookieServ...	0b154678e69d2141...	[[2, 2], [374, 374], [375, 37...	[{"Name": "nsListIter", "Sig...	[{"Name": "nsCookieAtt...	81d0f2e5c005a7...	[[2, 2], [374, 374], [37...	[{"Name": "nsListIter", "...	[{"Name": "nsCookieAttrib...
netwerk/cookie/src/nsCookieServ...	0b154678e69d2141...	[[43, 43], [57, 57], [152, 15...	[{"Name": "nsCookieEntry...	[{"Name": "nsCookieEn...	81d0f2e5c005a7...	[[43, 43], [58, 60], [15...	[{"Name": "nsCookieEnt...	[{"Name": "nsCookieEntry"...
js/src/jsscan.c	8c43834c9ea191dc...	[[801, 802], [810, 810]]	[{"Name": "js_InitScanner"...	[{"Name": "keyword", "...	6e2a3be5dd3622...	[[801, 806], [814, 814]]	[{"Name": "js_InitScann...	[{"Name": "keyword", "Sig...
js/src/jsxml.c	8c43834c9ea191dc...	[[2302, 2302], [2310, 2310...			6e2a3be5dd3622...	[[2303, 2307], [2315, ...		
rdf/base/src/nsRDFXMLSerializer...	60c66ac6ab1e0cc2e...	[[1, 1], [1009, 1009]]	[{"Name": "Create", "Sign...	[{"Name": "QNameColl...	27881a2aea6b34...	[[2, 2], [1010, 1012]]	[{"Name": "Create", "Si...	[{"Name": "QNameCollect...
dom/src/base/nsDOMClassInfo.c...	4f07681075f23d85e...	[[3825, 3825], [6239, 6239...	[{"Name": "NS_DOMClassI...	[{"Name": "nsContractI...	d52a331d97d0fb...	[[3825, 3825], [6239, ...	[{"Name": "NS_DOMCla...	[{"Name": "nsContractIDM...
content/xul/document/src/nsXUL...	5b0d3052c613806f...	[[116, 116], [1380, 1380]]	[{"Name": "IsChromeURI", ...	[{"Name": "nsProxyLoa...	d8a5fa7ae3462d...	[[117, 117], [1382, 13...	[{"Name": "IsChromeUR...	[{"Name": "nsProxyLoadStr...
layout/svg/renderer/src/cairo/ns...	63ae18c53f1f3cbf2...	[[40, 40], [90, 90], [94, 94],...	[{"Name": "~nsSVGCairoS...	[{"Name": "nsSVGCairo...	c608cb0396a938...	[[41, 41], [91, 91], [96,...	[{"Name": "~nsSVGCair...	[{"Name": "nsSVGCairoSur...
js/src/jsxml.c	645e227d83cbc286...	[[2895, 2895]]			5ea1608db431cb...	[[2895, 2895]]		
security/manager/ssl/src/nsNSSC...	19d24e194f47d75a...	[[862, 864]]			01c86dfa0981f1f...	[[862, 867]]		
security/manager/ssl/src/nsNSSC...	19d24e194f47d75a...	[[353, 353], [425, 425], [1...			01c86dfa0981f1f...	[[354, 357], [430, 433]...		
js/src/jsexn.c	ac5036c596e3caa88...	[[92, 96], [98, 110], [112, 1...	[{"Name": "exn_destroyPri...	[{"Name": "JSExnPrivat...	0a5b6d0a39130d...	[[92, 93], [95, 129], [1...	[{"Name": "CopyErrorR...	[{"Name": "JSExnPrivate", "...
js/src/jsexn.c	1f91717b498c95eff...	[[164, 164]]	[{"Name": "CopyErrorRep...	[{"Name": "JSExnPrivat...	6e4070bafb9a3e...	[[164, 164]]	[{"Name": "CopyErrorR...	[{"Name": "JSExnPrivate", "...
js/src/jsexn.c	a5be8b14c845bc9b...	[[122, 122], [193, 194], [2...	[{"Name": "CopyErrorRep...	[{"Name": "JSExnPrivat...	ae4f07ab92fd52f...	[[122, 122], [193, 196]...	[{"Name": "CopyErrorR...	[{"Name": "JSExnPrivate", "...
toolkit/components/passwordm...	e4162bbfc7e3da7c...	[[1794, 1794], [1923, 1923...	[{"Name": "SignonDataEn...	[{"Name": "SignonData...	d6016cf254069d...	[[1795, 1798], [1927, ...	[{"Name": "SignonData...	[{"Name": "SignonDataEnt...
security/nss/lib/ssl/sslcon.c	01216d9af7cd8292...	[[40, 40], [1582, 1582], [1...	[{"Name": "ssl2_Construct...	[{"Name": "ssl2SpecsStr...	d0fdcbf71c07bf9...	[[40, 40], [1582, 1582]...	[{"Name": "ssl2_Constru...	[{"Name": "ssl2SpecsStr", "...
widget/src/gtk2/nsWindow.cpp	edeb7818458adb96...	[[961, 961], [977, 978]]	[{"Name": "nsWindow", "S...		56cbba6387bd61...	[[962, 970], [985, 985]]	[{"Name": "nsWindow", ...	
widget/src/windows/nsWindow.c...	edeb7818458adb96...	[[2789, 2789], [2796, 2797...	[{"Name": "IsCursorTranslu...	[{"Name": "OleRegister...	56cbba6387bd61...	[[2789, 2789], [2795, ...	[{"Name": "IsCursorTran...	[{"Name": "OleRegisterMg...
netwerk/base/src/nsSimpleURI.cpp	ef25ce1f54d1d9b41...	[[168, 168], [196, 196]]	[{"Name": "nsSimpleURI", ...		2f7c781cd24740...	[[168, 168], [196, 202]]	[{"Name": "nsSimpleUR...	
netwerk/base/src/nsStandardURL...	ef25ce1f54d1d9b41...	[[518, 518], [1429, 1429]]	[{"Name": "EncodeString",...		2f7c781cd24740...	[[519, 520], [1432, 14...	[{"Name": "EncodeStrin...	
netwerk/base/src/nsURLHelper.c...	ef25ce1f54d1d9b41...	[[510, 510], [513, 514]]	[{"Name": "InitGlobals", "S...		2f7c781cd24740...	[[510, 510], [513, 515]]	[{"Name": "InitGlobals",...	
netwerk/protocol/res/src/nsResPr...	a273bc07b0256547...	[[334, 334]]	[{"Name": "EnsureFile", "Si...		853f382728d811...	[[334, 335]]	[{"Name": "EnsureFile", ...	
toolkit/components/passwordm...	15d505874744198b...	[[94, 94], [130, 130], [147, ...	[{"Name": "SignonDataEn...	[{"Name": "SignonData...	c5b3f647658a1a...	[[95, 95], [132, 132], [...	[{"Name": "SignonData...	[{"Name": "SignonDataEnt...
toolkit/components/passwordm...	15d505874744198b...	[[60, 60], [154, 154]]		[{"Name": "nsPassword...	c5b3f647658a1a...	[[61, 61], [155, 157]]		[{"Name": "nsPasswordMa...
netwerk/protocol/ftp/src/nsFtpC...	8dd9ac0bf9ced5b8...	[[1294, 1294], [1295, 1295...	[{"Name": "CanReadCach...		75858478eb29f6...	[[1295, 1295], [1297, ...	[{"Name": "CanReadCa...	
netwerk/protocol/http/src/nsHtt...	1696a9350aed6d41...	[[340, 340], [342, 348], [3...	[{"Name": "nsHttpDigestA...	[{"Name": "nsHttpDige...	39801af7f6a4b8...	[[341, 350], [352, 365]...	[{"Name": "nsHttpDiges...	
netwerk/protocol/http/src/nsHtt...	1696a9350aed6d41...	[[111, 111]]		[{"Name": "nsHttpDige...	39801af7f6a4b8...	[[112, 115]]		[{"Name": "nsHttpDigestA...
js/src/jsobj.c	f1cc12e174c927ede...	[[2817, 2823]]	[{"Name": "js_SetProtoOrP...		a5ceb3aaa6aa3f...	[[2817, 2818]]	[{"Name": "js_SetProto...	

Figure 3.8: An example of the CSV file generated after finding the files affected by vulnerabilities from the Mozilla project.

File Path	Topological Index	Affected	Vulnerable	Commit Hash	Author Date	Changed Lin...	Affected Functions	Affected Classes	CVEs
xen/xsm/flask/ss/services.c	14	No	No	0fe53c4f279...	2014-03-25 14:23:57	[[2038, 2051]]			
xen/xsm/xsm_core.c	14	No	No	0fe53c4f279...	2014-03-25 14:23:57	[[46, 47], [55,...			
xen/xsm/xsm_policy.c	14	No	No	0fe53c4f279...	2014-03-25 14:23:57	[[21, 21], [22,...			
xen/arch/arm/traps.c	15	Yes	Yes	8cfc8e52067...	2014-04-21 10:16:16	[[1354, 1357...	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/arch/arm/traps.c	17	Yes	Yes	60f737636c2...	2014-04-14 18:01:20	[[76, 76], [14...	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/include/asm-arm/cpregs.h	17	Yes	Yes	60f737636c2...	2014-04-14 18:01:20	[[117, 117], [...			["CVE-2014-2915"]
xen/include/asm-arm/cpregs.h	18	Yes	No	5496c642535...	2014-04-14 19:37:16	[[118, 118], [...			["CVE-2014-2915"]
xen/include/asm-arm/processor.h	17	Yes	Yes	60f737636c2...	2014-04-14 18:01:20	[[86, 86], [93,...		[{"Name": "cpuinfo_arm",...	["CVE-2014-2915"]
xen/include/asm-arm/processor.h	18	Yes	No	5496c642535...	2014-04-14 19:37:16	[[87, 91], [98,...		[{"Name": "cpuinfo_arm",...	["CVE-2014-2915"]
xen/arch/arm/traps.c	19	Yes	Yes	5496c642535...	2014-04-14 19:37:16	[[85, 85]]	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/arch/arm/traps.c	21	Yes	Yes	cda12fa0cee...	2014-04-14 19:46:43	[[85, 85]]	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/arch/arm/traps.c	23	Yes	Yes	11faafd9d97...	2014-04-14 19:00:14	[[76, 76]]	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/include/asm-arm/cpregs.h	23	Yes	Yes	11faafd9d97...	2014-04-14 19:00:14	[[118, 118], [...			["CVE-2014-2915"]
xen/include/asm-arm/processor.h	23	Yes	Yes	11faafd9d97...	2014-04-14 19:00:14	[[91, 91]]		[{"Name": "cpuinfo_arm",...	["CVE-2014-2915"]
xen/arch/arm/traps.c	25	Yes	Yes	a0453db0c6...	2014-04-15 11:45:28	[[76, 76], [14...	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/arch/arm/traps.c	26	Yes	No	0b182202fef...	2014-04-15 13:06:42	[[77, 80], [14...	[{"Name": "check_stack...	[{"Name": "", "Signature":...	["CVE-2014-2915"]
xen/include/asm-arm/cpregs.h	25	Yes	Yes	a0453db0c6...	2014-04-15 11:45:28	[[117, 117], [...			["CVE-2014-2915"]
xen/include/asm-arm/cpregs.h	26	Yes	No	0b182202fef...	2014-04-15 13:06:42	[[118, 118], [...			["CVE-2014-2915"]
xen/include/asm-arm/processor.h	25	Yes	Yes	a0453db0c6...	2014-04-15 11:45:28	[[94, 94], [99,...		[{"Name": "cpuinfo_arm",...	["CVE-2014-2915"]
xen/include/asm-arm/processor.h	26	Yes	No	0b182202fef...	2014-04-15 13:06:42	[[95, 101], [1...		[{"Name": "cpuinfo_arm",...	["CVE-2014-2915"]
xen/include/asm-arm/sysregs.h	25	Yes	Yes	a0453db0c6...	2014-04-15 11:45:28	[[42, 42], [50,...			["CVE-2014-2915"]
xen/include/asm-arm/sysregs.h	26	Yes	No	0b182202fef...	2014-04-15 13:06:42	[[43, 67], [76,...			["CVE-2014-2915"]
stubdom/grub/kexec.c	26	No	No	0b182202fef...	2014-04-15 13:06:42	[[70, 70], [13...			
stubdom/grub/mini-os.c	26	No	No	0b182202fef...	2014-04-15 13:06:42	[[738, 738], [...			
stubdom/grub/mini-os.h	26	No	No	0b182202fef...	2014-04-15 13:06:42	[[5, 5]]			

Figure 3.9: An example of the CSV file generated after building a topological timeline of every file in the Xen project.

3.4 Generating Security Alerts and Software Metrics

At this stage, we have the vulnerable and neutral versions of the files affected by CVEs that were reported for the five projects. As such, we are now able to use third-party tools to perform SCA in C/C++ code units (files, functions, classes), and extract security alerts and software metrics.

To generate alerts, the SATs Cppcheck[18] version 1.82 and Flawfinder[19] version 2.0.10 are used. These applications take the source files as their input and output their analysis in a

[18] `http://cppcheck.sourceforge.net/`
[19] `https://dwheeler.com/flawfinder/`

textual file format. This output is composed of alerts that highlight potential weaknesses or vulnerabilities, and which are located on a specific line of a source file. How these tools find different types of alerts is defined by a set of potentially dangerous patterns called rules. For the C/C++ programming languages, two rule examples include accessing uninitialized variables or calling known deprecated functions.

An example of an output file generated by the Flawfinder SAT is presented in Figure 3.10. Notice how the buffer overflow category is related to rules whose names are that of unsafe string manipulation functions, like `strcpy` or `sprintf`. Additionally, this SAT also displays any CWEs associated with a rule, mapping it to a vulnerability category.

By following the timeline created in Section 3.3, each file version could be easily retrieved and analyzed. Although both Cppcheck and Flawfinder were used to generate security alerts, time constraints only allowed the data from the latter to be used in the next sections.

File	Line	Level	Category	Name	Warning	Suggestion	CWEs
xpinstall/wizard/libxpnet/GUSI/include/GUSIDevice.h	301	5	race	chmod	This accepts filename arguments; if an attacker can mo...	Use fchmod( ) instead	CWE-362
xpinstall/wizard/libxpnet/GUSI/include/GUSIDevice.h	315	5	race	readlink	This accepts filename arguments; if an attacker can mo...	Reconsider approach	CWE-362, CWE-20
xpinstall/wizard/libxpnet/GUSI/include/GUSIMacFile.h	129	5	race	chmod	This accepts filename arguments; if an attacker can mo...	Use fchmod( ) instead	CWE-362
xpinstall/wizard/libxpnet/GUSI/include/GUSIMacFile.h	146	5	race	readlink	This accepts filename arguments; if an attacker can mo...	Reconsider approach	CWE-362, CWE-20
xpinstall/wizard/libxpnet/GUSI/include/sys/stat.h	105	5	race	chmod	This accepts filename arguments; if an attacker can mo...	Use fchmod( ) instead	CWE-362
xpinstall/wizard/libxpnet/GUSI/include/unistd.h	131	5	race	readlink	This accepts filename arguments; if an attacker can mo...	Reconsider approach	CWE-362, CWE-20
accessible/src/atk/nsAccessibleWrap.cpp	430	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
browser/components/migration/src/nsINIParser.cpp	168	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
browser/components/migration/src/nsINIParser.cpp	173	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
browser/components/migration/src/nsOperaProfileM...	461	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
browser/components/migration/src/nsOperaProfileM...	469	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
browser/components/migration/src/nsOperaProfileM...	475	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
browser/components/migration/src/nsOperaProfileM...	476	4	buffer	sprintf	Does not check for buffer overflows (CWE-120)	Use sprintf_s, snprintf, or vsnprintf	CWE-120
calendar/libxpical/oeICalContainerImpl.cpp	170	4	format	sprintf	Potential format string problem (CWE-134)	Make format string constant	CWE-134
calendar/libxpical/oeICalContainerImpl.cpp	189	4	format	sprintf	Potential format string problem (CWE-134)	Make format string constant	CWE-134
calendar/libxpical/oeICalImpl.cpp	767	4	buffer	strcpy	Does not check for buffer overflows when copying to d...	Consider using snprintf, strcpy_s, or...	CWE-120
calendar/libxpical/oeICalImpl.cpp	770	4	buffer	strcpy	Does not check for buffer overflows when copying to d...	Consider using snprintf, strcpy_s, or...	CWE-120
caps/src/nsScriptSecurityManager.cpp	1610	4	shell	system	This causes a new program to execute and is difficult to...	try using a library call that impleme...	CWE-78
caps/src/nsScriptSecurityManager.cpp	1617	4	shell	system	This causes a new program to execute and is difficult to...	try using a library call that impleme...	CWE-78
cck/cckwiz/ConfigEditor/mddriver.c	74	4	buffer	strcpy	Does not check for buffer overflows when copying to d...	Consider using snprintf, strcpy_s, or...	CWE-120

Figure 3.10: An example of the Flawfinder tool's output after analyzing the source code from the Mozilla project.

To generate software metrics, we use a single tool called Understand[20] version 4.0.837. The reason this application was chosen is that it was used to collect the metrics for any vulnerabilities reported between the years 2000 and 2016 in the work by Alves et al. [5]. In order to maintain the dataset's integrity, we must also use it in our work.

Understand is capable of generating over seventy metrics [28] for C/C++ code, including the ones presented in Section 2.3: Lines of Code (LOC), the Chidamber and Kemerer (CK) suite, Fan-In (FANIN), Fan-Out (FANOUT), and variants of McCabe's Cyclomatic Complexity. To see the full list of metrics used in our work along with a detailed description of each one, refer to Appendix 6.

This tool generally groups metrics into three categories[21] based on their nature. These are complexity metrics (e.g. cyclomatic complexity), volume metrics (e.g. LOC, FANIN, FANOUT), and object-oriented metrics (e.g. the CK suite). Moreover, Understand aggregates metrics in micro-level code units (functions and classes) to create metrics for entire files at a macro-level. The tool accomplishes this by using the average, sum, and maximum functions. Because some metrics in the original dataset were aggregated by Alves et al. [5] and not by Understand, we also had to emulate this behavior by adding more information to the output files.

Much like with security alerts, these software metrics were generated by following the file timeline. An example of Understand's output is shown in Figure 3.11. Notice how some cells are empty as certain metrics only apply to specific code units and must be aggregated

[20] `https://www.scitools.com/`

[21] `https://support.scitools.com/support/solutions/articles/70000582289-metrics-overview`

to extend them to others (e.g. the Cyclomatic column in functions and the AvgCyclomatic column in files). Note also that the struct, union, and class C/C++ constructs are all considered to be part of the "classes" category.

Kind	Name	File	CountLine	AvgLineCode	Cyclomatic	AvgCyclomatic	MaxCyclomatic	SumCyclomatic
File	32bitbios.c	tools/firmware/rombios/32bit/32bitbios.c	34	0		0	0	0
File	32bitbios_support.c	tools/firmware/hvmloader/32bitbios_support.c	160	40		9	17	18
File	32bitgateway.c	tools/firmware/rombios/32bitgateway.c	177	0		0	0	0
File	32bitprotos.h	tools/firmware/rombios/32bitprotos.h	16	0		0	0	0
File	8250-uart.h	xen/include/xen/8250-uart.h	157	0		0	0	0
Static Function	ADD_REGION(void *,unsigned long,struct xmem_pool *)	xen/common/xmalloc_tlsf.c	19		1			
Function	AES_cbc_encrypt(const unsigned char *,unsigned char *,...	tools/blktap2/drivers/aes.c	50		13			
Function	AES_cbc_encrypt(const unsigned char *,unsigned char *,...	tools/blktap/drivers/aes.c	50		13			
Function	AES_decrypt(const unsigned char *,unsigned char *,cons...	tools/blktap/drivers/aes.c	186		5			
Function	AES_decrypt(const unsigned char *,unsigned char *,cons...	tools/blktap2/drivers/aes.c	186		5			
Function	AES_encrypt(const unsigned char *,unsigned char *,cons...	tools/blktap/drivers/aes.c	186		5			
Function	AES_encrypt(const unsigned char *,unsigned char *,cons...	tools/blktap2/drivers/aes.c	186		5			
Function	AES_set_decrypt_key(const unsigned char *,const int,AE...	tools/blktap2/drivers/aes.c	47		4			
Function	AES_set_decrypt_key(const unsigned char *,const int,AE...	tools/blktap/drivers/aes.c	47		4			
Function	AES_set_encrypt_key(const unsigned char *,const int,AES...	tools/blktap/drivers/aes.c	97		14			
Function	AES_set_encrypt_key(const unsigned char *,const int,AES...	tools/blktap2/drivers/aes.c	97		14			
Function	APIC_init_uniprocessor()	xen/arch/x86/apic.c	49		5			
Function	ASSERT_NOT_IN_ATOMIC()	xen/common/preempt.c	6		10			
Struct	AddrRangeDesc	tools/libfsimage/zfs/mb_info.h	9	0		0	0	0
Struct	BDRVQcowState	tools/blktap/drivers/block-qcow2.c	51	0		0	0	0
Static Function	BLKTAP_MODE_VALID(unsigned long)	tools/blktap/lib/blktaplib.h	7		1			
Static Function	BLKTAP_MODE_VALID(unsigned long)	tools/blktap2/include/blktaplib.h	7		1			

Figure 3.11: An example of the Understand tool's output after analyzing the source code from the Xen project.

The process of generating both security alerts and software metrics is represented by the "Extract Alerts and Software Metrics" action in the right and middle sides of the diagram in Figure 3.1.

3.5 Storing the Collected Data in a Database

After collecting the vulnerability metadata and generating the security alerts and software metrics before and after each patch, this data was stored on disk to more easily relate and analyze it. To accomplish this, the authors of the original dataset designed a relational database where entities like vulnerabilities, code units, and patch information are related to one another based on their attributes. We inherit this design and augment it with additional fields and tables so as to store other types of data, namely the previously generated security alerts, SAT properties, vulnerability categories, and CWEs. The original dataset is stored in a MySQL[22] database, though the specific Database Management System (DBMS) is not particularly important as the resulting data can be migrated or exported to a different format. As such, we also use this DBMS to insert and update vulnerabilities, security alerts, and software metrics. More specifically, we used MySQL versions 8.0.22 (on Windows) and 8.0.26 (on Ubuntu).

An Entity–Relationship (ER) diagram of the original database is presented in Figure 3.12. What follows is a brief description of the most relevant tables in this database's schema, as well as the data we inserted or updated in each one:

- **VULNERABILITIES**: stores the vulnerabilities collected in Section 3.2 which are identified by their CVEs. Other information includes their publish date, CVSS score, how it impacts different security properties, vulnerability types, CWE identifier, and any URLs that link to the project's bug tracker platform. The original table structure was changed to include the vulnerability's CWE;
- **PATCHES**: stores any information regarding the neutral and vulnerable commits found in Section 3.3, i.e., a vulnerability's patch. This includes the Git commit

[22] `https://www.mysql.com/`

hash where a vulnerability was patched (neutral), the commit hash immediately before (vulnerable), the commit's author date and tag name, and a reference to any associated CVEs from the VULNERABILITIES table;

- **FILES_*, FUNCTIONS_*, and CLASSES_***: various tables that store the metadata and software metrics of code units in a project, combining the information collected in Section 3.3 with the metrics generated in Section 3.4. While the original database schema specified distinct tables for different modules within a project (e.g. FILES_1_javascript and FUNCTIONS_2_kernel, where the identifiers 1 and 2 represent the Mozilla and Linux Kernel projects), it was more practical to merge these for each pair of project and code unit kind (using the previous example, this would be FILES_1 and FUNCTIONS_2). Note that these tables represent code unit *versions*, rather than unique files, functions, or classes in a repository. As such, each record may be associated with one or more rows from the PATCHES table;
- **EXTRA_TIME_FILES, *_FUNCTIONS, and *_CLASS**: each vulnerability is associated with specific code unit versions by using the PATCHES table, which maps CVEs to the commit hashes. To then map this commit hash with a specific primary key in the FILES_*, FUNCTIONS_*, and CLASSES_* tables, the intermediary tables known as EXTRA_TIME_FILES, EXTRA_TIME_FUNCTIONS, and EXTRA_TIME_CLASS are used. These tables exist due to the high volume of files in the projects' repositories, and are used to facilitate the process of mapping a vulnerability or patch to a particular code unit version by speeding up the search query's execution time;
- **REPOSITORIES_SAMPLE**: a smaller table that lists each project, along with its numeric identifier and a hyperlink to its repository.

In order to store data relative to security alerts and SATs, it was necessary to update the original design with new tables. Figure 3.13 presents these additions in the form of an ER diagram. These newly added tables are described below.

- **SAT**: stores the name and identifier of each SAT. For our work, only Cppcheck and Flawfinder were inserted;
- **RULE**: stores each SAT's rules, including their names and the vulnerability category derived from its associated CWEs. This table does not contain an exhaustive list of every rule as they are only added as new alerts are inserted;
- **ALERT**: stores any security alerts generated by SATs in Section 3.4, including their location in a source file, level of severity, a warning message emitted by the tool, and which rule was responsible for generating it;
- **ALERT_FUNCTION and ALERT_CLASS**: intermediate tables that map an alert to one or more functions or classes, therefore listing which alerts also appear in these micro-level code units. This is accomplished by checking for any overlap between the alert's line number and the function and class ranges found in Section 3.3;
- **VULNERABILITY_CATEGORY**: stores a set of vulnerability categories that may be associated with zero or more CWEs;
- **CWE_INFO**: stores any CWEs associated with the CVEs in VULNERABILITIES or the categories in VULNERABILITY_CATEGORY. The complete list of

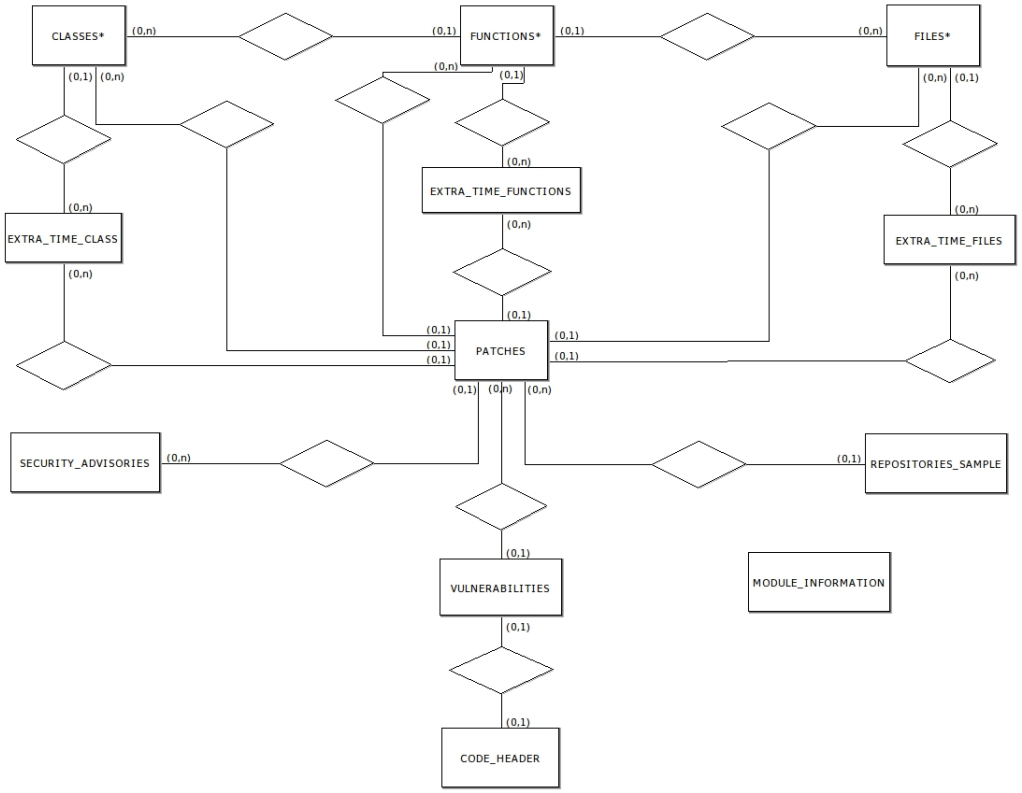


Figure 3.12: The ER diagram of the original database as designed by Alves et al. Publicly available in [7].

categories and their CWEs used for this work were adapted from Pereira and Vieira [40] and are shown in Table 3.3;

- **RULE_CWE_INFO**: an intermediate table that maps a SAT's rule to a CWE, and thus a vulnerability category.

Due to time constraints during this work's development, only security alerts generated by Flawfinder are considered. Even though Cppcheck was also used to emit alerts, the insertion process could not be completed for the Linux Kernel project in a timely fashion. Moreover, only around 56% and 73% of Flawfinder's output files were inserted for the Linux Kernel and Mozilla projects, respectively. No security alerts were generated for the Glibc project, though its software metrics were collected and inserted into the database.

With these new additions to the database's schema, we can now easily export a dataset comprised of every code unit's software metrics, security alert occurrences, vulnerability category, and whether or not it was affected by a CVE. The act of storing the collected vulnerability, security alert, and software metric information in a database is represented by the "Database" icon in the middle of the diagram in Figure 3.1.

3.6 Creating and Validating the Dataset

Finally, we reach a stage where we can transform the vulnerabilities, security alerts, and software metrics into a dataset capable of being fed into machine learning algorithms. For this work, we built classifiers that take a code unit (file, function, class) as their input,

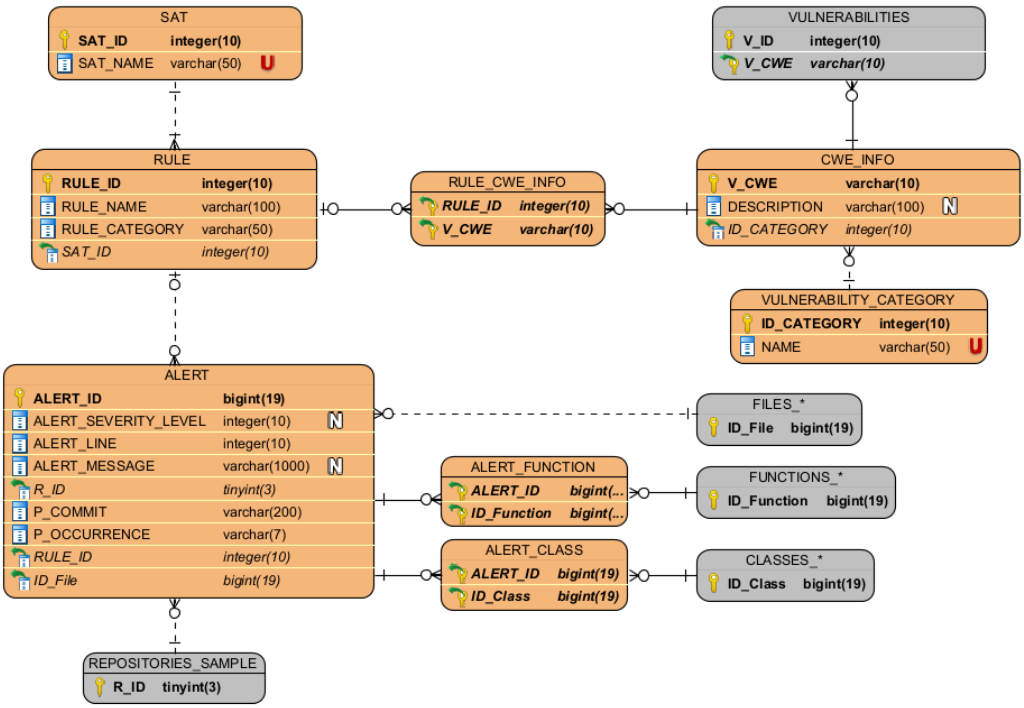


Figure 3.13: The ER diagram designed for the tables that store SAT information, their rules, generated security alerts, and descriptions for each CWE. Any tables that existed in the original schema are shown in grey and only with the relevant columns.

and output whether that unit is vulnerable or not. The final dataset is composed of many code units (samples) whose attributes (features) are used by the trained model to classify them.

These features belong to two categories: 1) values that count the number of times that a given SAT rule was responsible for generating a security alert in a code unit; 2) software metrics of the code unit in question. As we saw in 3.4, these metrics can sometimes only exist for a given code construct. With that being the case, we created three different datasets for detecting vulnerable files, functions, and classes. Due to time constraints, however, we will focus solely on training classifiers using the function dataset.

Much like the work of Pereira et al. [41] described in Section 2.5, we considered two types of target classes: 1) a binary label, where the code unit is either neutral or vulnerable (regardless of its vulnerability category); 2) a multiclass label, where the code unit is either neutral, vulnerable without a category, vulnerable in one of the categories specified in Table 3.3, or part of a special class that groups specific categories with fewer than 10% of vulnerable samples into a single label. We refer to this last class as the grouped multiclass label, whose purpose is to divide code units into vulnerability categories but also improve the machine learning results by aggregating categories that fall under a given sample threshold.

We create this dataset by enumerating every function and software metrics in the FUNCTIONS_* tables, while counting the occurrences of security alerts by their SAT rules via the SAT, RULE, ALERT, and ALERT_FUNCTION tables. To map a function to a vulnerability category we first find the CVE associated with its file's commit using the EXTRA_TIME_FILES, PATCHES, and VULNERABILITIES tables. Because each vulnerability may be associated with a CWE, we can use the CWE_INFO and VULNERA-

Category	CWEs
Memory Management	119, 362, 399, 416, 476, 824
Input Validation	20, 78, 79, 91, 94, 134, 189
Permission	255, 264, 269, 284, 287, 352
Data Protection	199, 200
Coding Practices	17, 19, 254
Cryptography	310
System Configuration	16
File Management	22, 59
Output Encoding	-
Error Handling and Logging	-
Communication Security	-
Database Security	-

Table 3.3: The vulnerability categories considered for this work and their respective CWEs. Adapted from [40].

BILITY_CATEGORY tables to determine the category or lack thereof. Only the Mozilla, Linux Kernel, and Xen projects were used to build the dataset as Glibc does not have any alerts associated with it while Apache has very few to be considered significant.

In order for the dataset to have some internal consistency regarding its labels, this data export process excluded two kinds of samples. First, any functions or classes whose lines numbers could not be determined in Section 3.3 are skipped since it would otherwise be impossible to associate them with any security alerts. Secondly, any code units whose commit had not yet gone through the alert insertion process were also excluded. This was done because not every Flawfinder output file (generated per commit) could be inserted into the database, as explained in Section 3.5.

An example of the final function dataset used experimentally can be seen in Figure 3.14. A function is considered neutral if its label is zero and vulnerable otherwise. Note how the features shown between the vulnerability category and labels represent some of the software metrics described in Section 2.3 (FANIN, LOC, FANOUT, McCabe's Cyclomatic Complexity), and the number of security alerts generated by the Flawfinder SAT. More specifically, these alerts were emitted based on the rules regarding the unsafe usage of the C functions `strcpy`, `strcat`, `sprintf`, and `memcpy`. Note also how the vulnerability category column is always mapped to the same multiclass label for vulnerable samples. This column was used as an intermediate value when computing the grouped multiclass label, where any category whose number of samples was too small was assigned a different value (in this case, the value 14).

Before validating the dataset by running machine learning experiments, we applied some preprocessing operations to improve the trained models. To start with, the dataset is highly imbalanced since 99.1% of the samples are neutral. Therefore, 79% of these were removed to lower this percentage to 95%, allowing the vulnerable labels to occupy the remaining 5%. Additionally, the creation of the grouped multiclass label mentioned above was done by assigning a different class to any vulnerability category with fewer than 10% of the

Description	VULNERABILITY_YEAR	VULNERABILITY_CATEGORY	CountInput	CountLine	CountOutput	Cyclomatic	Flawfinder_strcpy	Flawfinder_strcat	Flawfinder_sprintf	Flawfinder_memcpy	binary_label	multiclass_label	grouped_multiclass_label
1_45187...	2010	Input Validation	6	21	7	5	0	0	0	310	0	0	0
1_3efd4...	2009	Permission	3	7	5	1	59	118	0	0	0	0	0
1_7bac0...	2010	Input Validation	6	54	6	13	0	0	0	195	0	0	0
1_75b14...	2009	Memory Management	8	180	13	50	0	0	0	124	1	2	2
1_097c0...	2011	Input Validation	7	43	17	6	15	15	0	0	0	0	0
1_3efd4...	2009	Permission	7	29	4	4	0	0	0	0	0	0	0
1_e338c...	2014	Input Validation	17	1447	80	227	0	0	0	0	0	0	0
1_e338c...	2014	Input Validation	17	1466	80	230	0	0	0	0	1	3	14
1_3a110...	2016	Other	4	151	4	38	0	0	34	0	0	0	0
3_6e97c...	2015	Memory Management	11	57	6	15	0	0	0	0	0	0	0
1_75b14...	2009	Memory Management	28	197	12	26	0	0	10	0	1	2	2
1_a42bc...	2009	Memory Management	28	197	12	26	0	0	10	0	0	0	0
1_f8bf0a...	2012	Memory Management	5	27	4	4	2	0	1	0	0	0	0
1_bbff1a...	2017	Other	4	140	27	23	0	0	0	0	1	1	1
1_a0d1b...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_b048d...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_95bf1...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_11139...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_03d7a...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_0e46a...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_2842b...	2015	Input Validation	1	14	17	1	0	0	4	6	1	3	14
1_75b14...	2009	Memory Management	13	46	4	7	0	0	0	0	1	2	2

Figure 3.14: An example of the function dataset containing the software metrics and security alert occurrences for the Mozilla, Linux Kernel, and Xen projects.

vulnerable samples. This step affected every category in Table 3.3 with the exception of memory management. In the end, the function dataset was left with 85,915 total samples and four grouped multiclass values, with the following class distributions: 94.9% neutral, 2.2% vulnerable without a category, 1.7% vulnerable in the memory management category, and 1.2% vulnerable in the remaining categories. For the binary label, this would translate into 94.9% neutral and 5.1% vulnerable samples.

The process of creating datasets for each code unit is represented by the "Create Datasets" action in the middle left corner of the diagram in Figure 3.1.

Having applied all of the previously mentioned operations, we have now arrived at the final function dataset which we subject to machine learning experiments in Chapter 4. Here, we show the experimental results of training and testing classifiers with different kinds of parameters. Moreover, we answer specific Research Questions (RQs) while identifying any threats to validity that might have hindered our work. In particular, we explore how the year these vulnerabilities were published affects the models' performance metrics, and thus if this temporal data may be used to detect vulnerable function code.

Chapter 4

Validating the Vulnerable Function Dataset

This chapter describes the experimental steps followed in order to validate the function dataset developed in Chapter 3. The previously collected security alerts and software metrics are used as features when creating machine learning models capable of detecting vulnerable function code. A first stage is used to explore different classifier parameter configurations so as to determine which combination best excels at this task. After determining the most apt parameters, a temporal sliding window is applied to the data partitioning step in order to check if we can predict whether a function is vulnerable or not using data from previous years.

4.1 Overview

After completing the process outlined in Chapter 3 and building the function dataset we move onto the validation stage, where it will be used experimentally to train and test machine learning models so as to assess its quality in regards to detecting vulnerable code. This stage was divided into two main steps: 1) trying out various combinations of machine learning techniques and parameters in order to find the best performing configurations; 2) selecting the best configurations and using them to create classifiers trained on a version of the dataset that has been partitioned according to each vulnerability's year.

More concretely, these previous steps are used to answer the following Research Questions (RQs):

- **RQ1**: Can we use Static Analysis Tool (SAT) alerts and software metrics to predict vulnerable functions?
- **RQ2**: Can we predict whether a function is vulnerable or not using static data from previous years?

For the former step, we used the Propheticus[1] tool version 2021-03-03 [11] mentioned in Section 2.5 to automate this process. In order to train the models, we first had to define which configurations of classification algorithms, dimensionality reduction techniques, data balancing methods, and target labels to use. Because we want to explore the highest

[1] `http://www.joaorcampos.com/ISSRE-2019/`

possible number of configurations, we used every combination of the techniques shown in Table 4.1 with the algorithm hyperparameters specified in Table 4.2. This resulted in a total of 848 configurations, or 424 for both the binary and grouped multiclass labels. All of these values were used by Pereira et al. [41], though we chose a smaller subset due to time constraints. Internally, Propheticus used stratified 5-fold cross-validation, where the training and testing data is randomly divided into five subsets (four for training and one for testing) with the same class distribution as the dataset [50]. This specific number of folds was chosen to lower the execution time. For this same reason, we also only ran each configuration five times.

Parameter	Values
Target Labels	Binary, Grouped Multiclass
Algorithms	Random Forest (RF), Bagging, Extreme Gradient Boosting (XGB)
Dimensionality Reduction	Variance, Variance and Correlation
Data Balancing	None, Random Undersampling, Random Oversampling, Random Undersampling and Random Oversampling

Table 4.1: The target labels, classification algorithms, dimensionality reduction techniques, and data balancing methods used to validate the function dataset in Propheticus.

Algorithm	Hyperparameters
RF	**n_estimators**: [100, 200], **criterion**: [gini], **min_samples_split**: [0.001, 2], **min_samples_leaf**: [0.001, 1], **max_features**: [none], **bootstrap**: [true]
Bagging	**n_estimators**: [100, 200], **max_features**: [0.1, 0.55, 1.0], **bootstrap**: [true]
XGB	**n_estimators**: [100, 300], **learning_rate**: [0.1, 0.3], **gamma**: [0], **subsample**: [1], **max_depth**: [10, 30], **min_samples_split**: [2, 5], **min_samples_leaf**: [1, 4]

Table 4.2: The hyperparameters of the classification algorithms used to validate the function dataset in Propheticus.

This process is represented by the "Train and Test Classifiers Using Different Configurations" action on the left side of the diagram in Figure 3.1. After doing so, we arrive at the following basic performance metrics for each configuration:

- **True Positive (TP)**: a code unit was correctly classified as vulnerable;
- **False Positive (FP)**: a neutral code unit was incorrectly classified as vulnerable;
- **True Negative (TN)**: a code unit was correctly classified as neutral;
- **False Negative (FN)**: a vulnerable code unit was incorrectly classified as neutral.

These can in turn be used to compute performance metrics that were seen in the majority of the related work in Sections 2.4 and 2.5. They include precision, recall, and F-score [50], and are described in Table 4.3. In order to apply these metrics to classes with more than two labels, they are aggregated using a weighted average which takes into account the proportion of each label [24].

By comparing the precision, recall, and F-score metrics for every configuration executed by Propheticus, we can select the best performing combination of parameters per label and

Performance Metric	Formula	Description
Precision	$\frac{TP}{TP+FP}$	Percentage of cases classified as positive that are truly positive
Recall	$\frac{TP}{TP+FN}$	Percentage of positives cases that are correctly classified
F-score	$\frac{2 \times precision \times recall}{precision+recall}$	Harmonic mean of the precision and recall metrics

Table 4.3: Traditional performance metrics for assessing a classifier's quality.

metric. We take this opportunity to explore some other questions regarding the detection of vulnerable function code. More concretely, we tackle **RQ2** by checking whether data from previous years can be used to predict if a function is vulnerable or not. Instead of letting Propheticus partition the data randomly, we compose each training and testing subset such that the first one contains vulnerabilities that were detected before the ones in the testing set.

As can be seen in the dataset in Figure 3.14, each function is associated with the year when its vulnerability was published given that this value may be extracted from the Common Vulnerabilities and Exposure (CVE) identifier. To perform such a data split, we make use of what we called a temporal sliding window. The years chosen for the training set are determined by the size of this window, while any testing samples are retrieved from the single year immediately after. This window starts at a given year and "slides" until we reach 2019 as the testing year. For example, a five-year window beginning in 2008 would result in the following list of tuples, where the first element represents the range of training years, and the second the testing year: (2008-2012, 2013), (2009-2013, 2014), and so on until (2014-2018, 2019). A visual representation of this temporal partitioning can be seen in Figure 4.1.

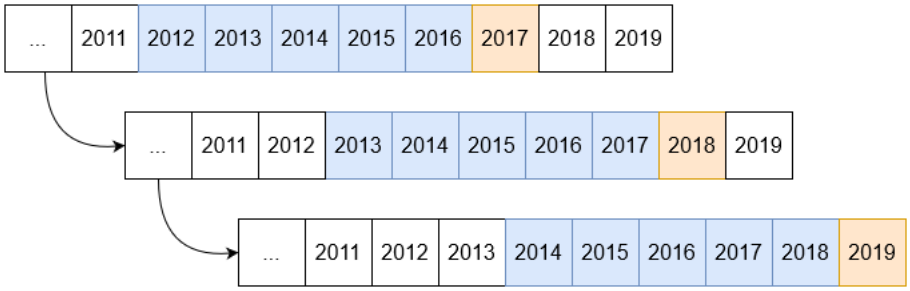


Figure 4.1: A diagram showing the temporal sliding window dividing the training (in blue) and testing (in orange) function data for several years until 2019. using a window size of five years.

The total number of samples per vulnerability year can be seen in Table 4.4. Note that, although one of the main goals in Chapter 3 was to update the database with data from 2016 to 2019, our automated process also collected vulnerabilities from earlier years that were not scraped by Alves et al. [5]. Due to the low amount of samples in the earlier years, we considered 2013 as the first testing year. Additionally, three window sizes were chosen: five years, ten years, and a variable-length window where every previous year is used for training. Given that we only reran the best configurations, the number of executions per classifier was increased to thirty.

This process is represented by the "Choose the Best Configurations", "Partition the Functions Using a Temporal Sliding Window", and "Train and Test Classifiers Using the Best Configurations" actions at the bottom of the diagram in Figure 3.1.

Vulnerability Year	Number of Samples
2002	95
2003	0
2004	47
2005	308
2006	353
2007	141
2008	466
2009	3074
2010	1243
2011	3745
2012	8455
2013	10292
2014	15845
2015	12495
2016	16357
2017	6838
2018	3014
2019	3147

Table 4.4: The number of samples in the function dataset in each year, as determined by a vulnerability's CVE identifier.

4.2 Research Question 1: Exploratory Experiments

In Section 4.1, we outlined the classification algorithms, hyperparameters, and data preprocessing techniques that were combined to form 424 different classifier configurations per target label. Similar to other authors [11, 41], the process of using them to train and test machine learning models capable of identifying vulnerable functions was automated using the Propheticus tool. This section will present the obtained results and analyze them so as to answer **RQ1**.

A list describing the best configuration parameters for each performance metric and target label mentioned in Section 4.1 is presented in Table 4.5, with the highest values for each metric shown in Table 4.6. In total, nine configurations were found to yield the best performance metrics, with three pairs referring to the same label and metric (C_3 and C_4, C_6 and C_7, C_8 and C_9).

In all cases, variance threshold appears as the dimensionality reduction technique used by the best configurations. The final function dataset consisted of 154 features, with 28 being software metrics and 126 the number of SAT alert occurrences. With this method, any feature with a variance under a specific threshold (zero for the Propheticus tool) would be removed from the training and testing data [50]. In other words, this meant that any column that always had the same value was discarded. In practice, this only applied to features representing the Flawfinder SAT's rules. This can be justified by the fact that, while these additional rules existed in the database, the considered function samples did not contain any alert associated with them. After applying this feature selection technique, 74 features were discarded leaving only 52 alert rules and a total of 80 features in the dataset.

Name	Label	Algorithm	Data Balancing	Dim. Reduc.	Hyperparameters
C_1	Binary	RF	Undersampling	Variance	**bootstrap**: true, **criterion**: gini, **max_features**: auto, **min_samples_leaf**: 1, **min_samples_split**: 2, **n_estimators**: 100
C_2	Binary	Bagging	None	Variance	**bootstrap**: true, **max_features**: 0.55, **n_estimators**: 200
C_3	Binary	Bagging	None	Variance	**bootstrap**: true, **max_features**: 1.0, **n_estimators**: 100
C_4	Binary	Bagging	None	Variance	**bootstrap**: true, **max_features**: 1.0, **n_estimators**: 10
C_5	Multiclass	RF	Undersampling	Variance	**bootstrap**: true, **criterion**: gini, **max_features**: auto, **min_samples_leaf**: 1, **min_samples_split**: 2, **n_estimators**: 100
C_6	Multiclass	Bagging	None	Variance	**bootstrap**: true, **max_features**: 0.55, **n_estimators**: 100
C_7	Multiclass	Bagging	None	Variance	**bootstrap**: true, **max_features**: 0.55, **n_estimators**: 200
C_8	Multiclass	RF	None	Variance	**bootstrap**: true, **criterion**: gini, **max_features**: none, **min_samples_leaf**: 1, **min_samples_split**: 2, **n_estimators**: 200
C_9	Multiclass	Bagging	None	Variance	**bootstrap**: true, **max_features**: 1.0, **n_estimators**: 200

Table 4.5: A description of the machine learning parameter configurations that yielded the best results for each target label and performance metric.

Similarly, it is interesting to note that, despite the dataset only having 85,915 samples, the data balancing techniques associated with the best configuration either removed data (with random undersampling) or did nothing to the class distribution.

To get a better understanding of the performance metrics presented in Table 4.6, let us consider what they would mean in practice when detecting vulnerable code. As mentioned in Section 3.1, a mechanism such as a trained classifier would in useful when analyzing a large codebase, where searching for vulnerabilities expends both time and resources. One may consider, for example, a plugin in an Integrated Development Environment (IDE) that reports potentially vulnerable functions that should be revised by a person. Here, the higher the recall, the more relevant functions would be marked as potentially vulnerable. Similarly, the higher the precision, the more these marked functions would actually be vulnerable. To measure how both of these evolve at once, the harmonic mean given by the F-score is used.

In terms of precision (C_1, C_5), the best results for both binary and grouped multiclass

Name	Description	Precision	Recall	F-score
C_1	Best Precision (Binary)	**0.9373**	0.6935	0.7796
C_2	Best Recall (Binary)	0.9369	**0.9511**	0.9383
C_3	Best F-score (Binary)	0.9366	0.9503	**0.9394**
C_4	Best F-score (Binary)	0.9366	0.9503	**0.9394**
C_5	Best Precision (Multiclass)	**0.9362**	0.5906	0.7095
C_6	Best Recall (Multiclass)	0.9342	**0.9505**	0.9367
C_7	Best Recall (Multiclass)	0.9341	**0.9505**	0.9367
C_8	Best F-score (Multiclass)	0.9344	0.9500	**0.9379**
C_9	Best F-score (Multiclass)	0.9344	0.9500	**0.9379**

Table 4.6: The best results for the precision, recall, and F-score performance metrics for each configuration described in Table 4.5, rounded to four decimal places. The relevant metric value for each configuration is shown in bold.

labels (0.9373 and 0.9362, respectively) were observed for the RF algorithm using random undersampling, where the samples from the majority class (i.e. neutral functions) are discarded until they reach the proportions of the second majority label [36]. Conversely, the recall for both configurations suffered a significant drop in value, reaching only 0.6935 and 0.5906 for the binary and multiclass labels, respectively. Consequently, the F-score for these configurations is the worst on Table 4.6.

A confusion matrix showing how a classifier trained with C_5 predicted a function's vulnerability status during the testing phase is presented in Figure 4.2. The markers refer to the following four grouped multiclass labels: Memory Management (MM), Neutral (N), Vulnerable With No Category (V(NC)), and Vulnerable With A Category (V(WC)). Values that fall in the diagonal cells correspond to correct predictions. Note that this matrix is computed once for all testing results after performing the five runs. The confusion matrices for the remaining configurations are presented in Appendix 6.

Regarding the recall metric (C_2, C_6, C_7), similar results can be seen for both binary and multiclass labels. In this case, two configurations for the multiclass showed the exact same highest value, 0.9505, with the binary configuration being very close at 0.9511. Much like C_1 and C_5, these three configurations also share the same classification algorithm and and data balancing technique (Bagging and no sampling). On the other hand, unlike the recall in the C_1 and C_5, these configurations have a substantially higher precision value associated with them: 0.9369, 0.9342, and 0.9341 for C_2, C_6, and C_7. Note also that this precision is only a mere 0.004 and 0.020 apart from the best overall precision in C_1 and C_5, respectively. This shows that, for these three configurations, a compromise between precision and recall does not have to be made, allowing for a high value of approximately 0.94 for the F-score.

Finally, by taking a look at both precision and recall at once using the F-score metric (C_3, C_4, C_8, C_9), we can see equally high values. All four configurations used no data balancing, with every one but C_8 using the Bagging classification algorithm. The values for each pair, 0.9394 and 0.9379, respectively, are both close to each other, while also being similar to the F-score values for the best recall configurations above.

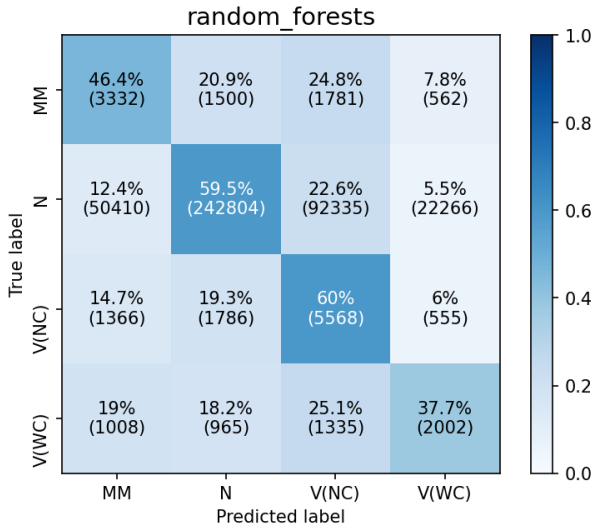


Figure 4.2: The confusion matrix showing the predictions of the classifier trained with configuration C_5.

Although the XGB classification algorithm was responsible for 512 of the 848 configurations, it did not appear once in the best results for any performance metric shown in Table 4.6. RFs and Bagging were considered for 128 and 96 configurations, respectively. Following the structure of this same table, the best results for XGB only appear at: the 22nd and 14th places (for a precision 0.9364 and 0.9353 in the binary and grouped multiclass labels), 4th place (for a recall of 0.9504 for both labels), and 8th place (for an F-score of 0.9391 for both labels).

Despite obtaining poor results for the recall and F-score in C_1 and C_5, the remaining best configurations show how one does not have to necessarily compromise precision for recall when detecting vulnerable function code. For these other configurations, the F-score was greater than 93% for cases when functions had to be classified as either vulnerable or neutral, and for cases when more vulnerability categories were at play. As such, we can answer **RQ1** in the affirmative given that we were able to use both software metrics and security alerts generated by SATs to predict vulnerable functions with high performance metrics. Moreover, particular emphasis seems to have been given to the RF and Bagging classification algorithms, along with the random undersampling or no sampling techniques, and the variance threshold feature selection method.

4.3 Research Question 2: Temporal Window Experiments

In Section 4.2, we explored 848 different machine learning configurations before arriving at the nine best configurations across both target labels and all three performance metrics. We are now able to focus solely on these parameters in order to analyze more specific situations regarding vulnerability detection. In particular, this section reruns the best configurations by first partitioning the training and testing subsets according to their year, so as to answer **RQ2**. Because Propheticus does not offer a way to split the data in a custom way, every classifier generated for this section was trained and tested using our scripts. For more technical details on these, refer to Appendix 6.

The results for the best binary classifier with regards to precision, C_1, are presented in Table 4.7, with a graphical representation of the same values in Figure 4.3. For brevity's

sake, the same tables and figures for the remaining configurations are shown in Appendix 6. Note, however, that C_4, C_7, and C_8 are omitted from this temporal analysis as they refer to duplicate pairs of target labels and performance metrics. Moreover, due to the limited number of years, a set of training years may be the same for different window sizes, meaning they would essentially represent the same results. In this case, a window size of ten years is the same as the variable-length window when using 2013 as the testing year. As such, this repeated year range is only listed under the "Variable" window in Table 4.7.

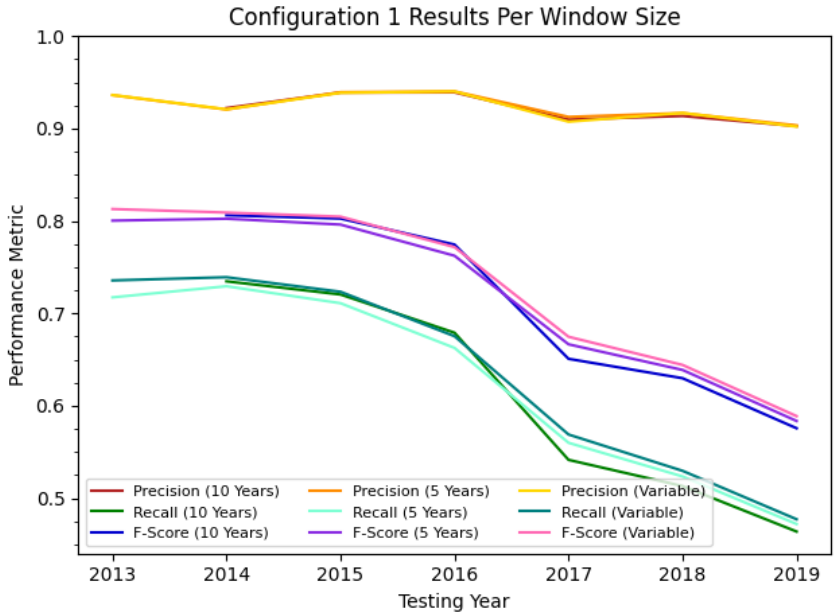


Figure 4.3: The evolution of the performance metrics for each window size along each testing year for configuration C_1.

By taking a look at Figure 4.3 and the remaining plots in the Appendix, we notice that the evolution for each performance metric is very similar for different window sizes. Although there exist nine lines in each plot (three metrics for three windows), we can consider them as only three line groups, each referring to a different performance metric. In practice, this means that, for the same configuration, the sliding window size has little impact.

All figures also show an interesting behavior in specific year ranges. When testing with function data between 2014 and 2016, there is an increase in performance for all metrics. These metrics then decrease from 2017 onwards, never reaching the previous high values. The only exception to this is for the precision metric in C_1 and C_5, where their values remain between 0.9 and 1.0. This can be explained by the fact that both of these configurations yielded the best precision in Section 4.2, while showing the lowest recall and F-score. As such, a trade-off between precision and recall is to be expected.

One possible explanation for the discrepancy in performance between 2014-2016 and 2017-2019 may be related to the training percentages shown in Table 4.7 and the remaining ones in the Appendix. Because of the vulnerability year distribution seen in 4.4, when the sliding window includes 2016 it increases the training subset from around 76% to 90% or more. This means that only 10% or fewer samples are used in the testing phase. Due to the rigid nature of this data partitioning step, techniques such as stratified cross-validation used by Propheticus in Section 4.2 can not be applied. In turn, this may mean that the classifier is left with a small testing subset that is not representative of the samples in the complete dataset. Although the experiments in this section were executed thirty times

Window	Training	Testing	Training %	Precision	Recall	F-score
10	2004-2013	2014	64%	0.9222	0.7347	0.8062
10	2005-2014	2015	78%	0.9391	0.7204	0.8027
10	2006-2015	2016	77%	0.9393	0.6791	0.7745
10	2007-2016	2017	91%	0.9098	0.5414	0.6508
10	2008-2017	2018	96%	0.9137	0.5130	0.6298
10	2009-2018	2019	96%	0.9027	0.4638	0.5756
5	2008-2012	2013	62%	0.9362	0.7173	0.8004
5	2009-2013	2014	63%	0.9207	0.7294	0.8025
5	2010-2014	2015	76%	0.9394	0.7110	0.7961
5	2011-2015	2016	76%	0.9403	0.6626	0.7624
5	2012-2016	2017	90%	0.9127	0.5599	0.6665
5	2013-2017	2018	95%	0.9169	0.5234	0.6387
5	2014-2018	2019	95%	0.9035	0.4719	0.5835
Variable	2002-2012	2013	64%	0.9364	0.7357	0.8129
Variable	2002-2013	2014	64%	0.9209	0.7391	0.8091
Variable	2002-2014	2015	78%	0.9387	0.7234	0.8047
Variable	2002-2015	2016	78%	0.9402	0.6753	0.7717
Variable	2002-2016	2017	91%	0.9075	0.5688	0.6747
Variable	2002-2017	2018	96%	0.9169	0.5295	0.6442
Variable	2002-2018	2019	96%	0.9022	0.4771	0.5887

Table 4.7: The performance metric values for configuration C_1 using the three temporal sliding windows.

for each configuration (as opposed to only five like in Section 4.2), the lack of a more sophisticated partitioning technique means that each run would yield very similar results.

Since the temporal sliding windows' effect seems to be minimal, we can try looking at the overall performance of each pair of configurations here as opposed to the one in Section 4.2. As mentioned before, both C_1 and C_5 offered good precision at the expense of recall, reaching the highest values of 0.9403 and 0.9387, respectively. Despite using the windows sizes of five and ten years, these best results were observed when 2016 function data was used for testing.

For C_2 and C_6 (best recall for binary and multiclass labels), the recall was higher than precision for all cases. Much like in Section 4.2, the trade-off between precision and recall was not as substantial as with C_1 and C_5, resulting in a better F-score. With that being said, the precision was lower than 0.90 in some cases. The best recall was 0.9564 for the window sizes of five and ten in C_2, and also 0.9564 for a five-year window in C_6. Like C_1 and C_5, this was observed for the 2016 testing year.

Finally, similar results can be seen for C_3 and C_9 (best F-score for binary and multiclass labels), where the F-score was over 0.9 (or very close to it) while prioritizing recall over precision. The highest value for C_9, 0.9358, was the same for all three window sizes using the 2016 data. The same is true of C_3, though only for a variable-length window with the value 0.9373.

Despite splitting the training and testing subsets according to their vulnerability years using a sliding window, the impact of this process seems to have been minimal. Firstly, the window size did not yield any substantially different results for each performance metric. Moreover, the strict nature of this partitioning appears to have hindered the

testing phase, as there was a big jump in the number of training samples when data from 2016 was included. The samples chosen for the testing data may have been few and not representative of the distribution in the complete dataset. In order words, this essentially meant that we applied a less robust mechanism than the one used in Section 4.2 at no greater benefit. As such, we cannot answer **RQ2** in the affirmative using this fixed year partitioning scheme, as it is not possible to conclude that we can predict whether a function is vulnerable or not given static data from earlier years.

4.4 Threats to Validity

This section will cover any weaknesses that might have influenced the results presented in Sections 4.2 and 4.3, and thus hindered the conclusions drawn for **RQ1** and **RQ2**. The following are some identified threats to this work's validity:

- Only one SAT, Flawfinder, was used to generate security alerts due to time constraints. Not only does this reduce the number of features significantly, but it also makes any created classifier heavily dependant on this tool's quality;
- All machine learning algorithms specified in Section 4.1 are based on decision trees [41]. Thus, better results may have been achieved had other types of classifiers been used in conjunction, such as the Support Vector Machines (SVMs) or neural networks mentioned in Sections 2.4 and 2.5;
- The function dataset only contained 85,915 samples, many of which may have been further removed when generating the classifiers due to the data balancing techniques specified in Section 4.1. In particular, the Glibc project was not used due to time constraints when generating alerts. By considering this project, the number of function samples would have increased;
- Due to the output format of Understand and the Clang compiler, it is very possible that many security alerts were not properly assigned to functions. Understand does not emit line numbers, meaning any code units had to be associated by name. Moreover, it was noted that Clang does not always include certain functions or classes in the Abstract Syntax Tree (AST) output. Put together this means that, while a SAT may have generated alerts for a function, the corresponding entry in the database might be missing its line numbers. Thus, vulnerable functions can become associated with zero alerts, when in fact they existed and could represent potential vulnerabilities. To combat the lack of line numbers in Understand, we computed a ratio that measured the difference between each function or class signature, allowing us to compare the strings reported by both tools. All code units over a specific ratio threshold were logged and reviewed. No outright incorrect cases were found during a testing session, though it is possible that some wrong values may have existed when considering the millions of code unit rows in the database.

In the next chapter, we will describe the work environment in more detail, while explaining certain key moments during development. This includes how version control was handled, how tasks were outlined, and how any major implementation roadblock was tackled.

Chapter 5

Work Execution

This chapter describes how the work presented in previous sections was conducted, including how its code and tasks were managed, what the work environment looked like, and any notable moments during development that required special attention.

This project began on September 14th, 2020, and ran until October 30th, 2021, the date when this very document was finalized and submitted for evaluation. During this time, a total of 63 weekly meetings took place between the student and the advisors. Due to the COVID-19 pandemic, the vast majority of these were held remotely using the Google Meet[1] and Zoom[2] video teleconferencing services. In order to improve communication, the instant messaging software Telegram[3] was used to quickly resolve any issues and iterate over the developed code.

All code created during this project, as well as any artifacts generated by our scripts, are kept in a GitHub repository[4]. This repository was shared with the advisors, allowing for any code to be reviewed before having it generate the final results. Various tasks (e.g. developing scripts used in Chapters 3 and 4, researching specific questions, writing parts of this document, etc) were represented as issues in GitHub. This allowed the student to adopt the following process: 1) develop the necessary code for each task in a separate Git branch; 2) create a pull request which would associate that code with a given issue; 3) have an advisor review it before merging the changes to the main branch. Some completed tasks can be seen in GitHub's issues tab presented in Figure 5.1. Notice how tasks are labeled according to different topics, such as static analysis or scraping data. The complete documentation of each script can be found in Appendix 6.

These developed scripts were executed in two different environments: the student's personal Windows 10 machine, and a remotely accessible Ubuntu 20.04 machine provided by the university. We will refer to these as the local and remote machines, respectively. For the majority of this project, the local machine was used to develop and test the scripts for small input data. After finalizing each script, these would then be left running anywhere from a few minutes to a few days in the remote machine. The progress and estimated time to completion were measured by using log files which would be periodically written to by each script. Towards the end of our work, the temporal window experiments described in Section 4.3 were executed solely on the local machine.

[1] `https://meet.google.com/`
[2] `https://zoom.us/`
[3] `https://web.telegram.org/`
[4] `https://github.com/joaohenggeler/uc-masters-software-vulnerabilities`

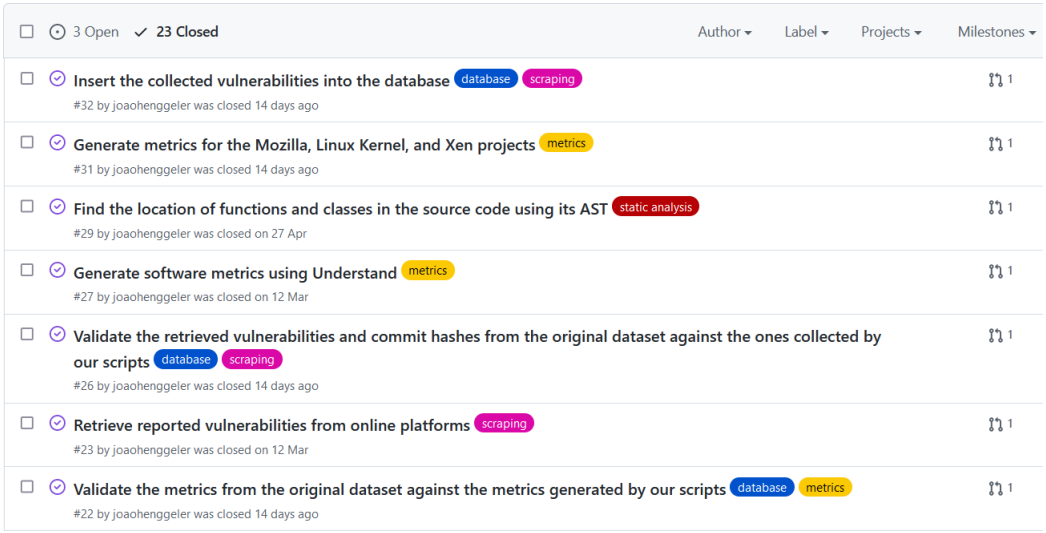


Figure 5.1: A screenshot of various tasks represented as issues in our GitHub repository.

One of the major tasks executed on the remote machine was the software metrics collection mechanism using the Understand tool, as described in Section 3.4. There, we mentioned that a specific version (4.0.837) was used by Alves et al. [5] to create the original database. Because part of our work revolved around extending this data, we too had to use this version. However, Understand version 4.0 could no longer be accessed with an academic license through SciTools' website[5]. As such, we contacted SciTools' support and were issued a 30-day promotional code to be used with older versions. This process had to be done twice so that our scripts could execute Understand for more than thirty days.

After generating software metrics using Understand 4.0, these were inserted in each C/C++ project's code unit table in the database. However, due to a property of the original database schema and the high amount of data generated for the Mozilla and Linux Kernel projects, a critical error occurred and prevented the MySQL database from being accessed correctly. The original schema used the INT data type (4 bytes) to store each primary key in the FILES_*, FUNCTIONS_*, CLASSES_*, EXTRA_TIME_FILES, EXTRA_TIMES_FUNCTIONS, and EXTRA_TIME_CLASS tables. Since the last two digits of this key are reserved for the project's identifier, its value would increment by one hundred with each inserted code unit. In turn, this meant that the primary key was exhausted after reaching the maximum signed 4-byte integer value (2147483647) in MySQL[6].

A conversion from the INT data type to BIGINT (8 bytes) was attempted but failed due to the large amounts of data present in an already populated EXTRA_TIMES_FUNCTIONS table. Additionally, this failure resulted in what was possibly a corrupted database, preventing any future reads or writes. Because of this, the only solution was to discard the database, change any relevant primary keys to the BIGINT data type, and reimport everything a second time. This entire process cost us about two weeks' worth of execution time in the remote machine, though this was not as severe as it could have been given that we had also saved any newly generated data in Comma-Separated Values (CSV) files on disk.

In the next chapter, we draw conclusions based on the validation experiments conducted and questions posed in Sections 4.2 and 4.3. Additionally, the contributions of this work are summarized, including the C/C++ function dataset composed of both software metrics

[5] `https://licensing.scitools.com/download`
[6] `https://dev.mysql.com/doc/refman/8.0/en/integer-types.html`

and security alerts whose build steps were described in Chapter 3.

Chapter 6

Conclusion

In this work, we presented a process through which one can build datasets of vulnerable code units along with data collected using Static Code Analysis (SCA). We explained how online platforms could be scraped for vulnerabilities reported for multiple products, as well as how we could then associate each one of them with specific file versions in a software project's version control system. An already existing but out-of-date database of vulnerabilities was updated with information from 2016 to 2019. We showed how tools such as Understand or the Clang compiler could be used to generate software metrics at a C/C++ file, function, and class level, while also locating these last two code units' line numbers. This data was then combined with security alerts from a Static Analysis Tool (SAT) called Flawfinder, allowing us to tally the number of potential vulnerabilities associated with each code unit.

We created three datasets for files, functions, and classes, each one composed of specific software metrics and enriched with security alerts related to potentially unsafe code patterns. Not only are these datasets capable of being fed to a wide array of machine learning algorithms, but we also built an automated process that may be used to collect future vulnerabilities in the years to come. Moreover, we made strategic improvements to the original database schema that resulted in a language-agnostic mechanism, leaving room for analyzing other programming languages in the future.

In order to assess the created datasets' quality, we focused on detecting vulnerable function code. We generated a dataset composed of 85,915 function samples, each being labeled one of two ways: 1) using a binary label, where a function may be neutral or vulnerable; 2) using a multiclass label, where a function may be neutral or belonging to one of several categories. In practice, we adapted this last label to the size of our dataset and created a grouped multiclass label with four possible values: neutral, vulnerable with no category, vulnerable in the memory management category, or vulnerable with a category with fewer than 10% of the samples. In total, our function dataset contained 80 relevant features, with 28 being software metrics and 52 being alert rules.

We posed two Research Questions (RQs) that served as a foundation for validating the function dataset using machine learning algorithms: 1) whether SAT alerts and software metrics could predict vulnerable functions; 2) whether we can predict if functions are vulnerable or not using static data from earlier years. An exploration of as many as possible machine learning parameter combinations took place, where we used the Propheticus tool to train and test classifiers using different configurations. A total of 848 configurations were tried, with the best ones demonstrating that it was indeed possible to predict vulnerable functions using metrics and alerts for both kinds of labels.

Results showed precision, recall, and F-score values as high as 93.7%, 95.1%, and 93.9%, respectively. Moreover, we saw that one did not have to trade off precision to still attain high recall values. The Random Forest (RF) and Bagging classification algorithms, as well as the variance threshold, random undersampling, and no sampling techniques stood above the rest in regards to performance.

When partitioning the dataset using a temporal sliding window in vulnerability years, we confirmed the previous performance metric values, though we were unable to adequately answer the second RQ. Results showed that splitting data using a window of five, ten, or every previous year had virtually no effect for the same performance metrics. In addition to this, we explained how this rigid data partitioning could lead to a testing subset that would be too small and not representative of the complete dataset for years between 2017 and 2019. In general, most metrics reached lower values than the best configurations in the exploratory experiments. Having said that, we saw how all performance metrics could still reach as high as 93% in certain scenarios.

As future work, other SATs could be used to generate security alerts and thus increase the number of features in the dataset, such as the Cppcheck tool whose output was omitted due to time constraints. In the same vein, projects in programming languages other than C and C++ could also be analyzed. Moreover, a more in-depth investigation of the influence of static data from previous years in code units could take place. This should be accompanied by a more robust data partitioning scheme that takes into account each subset's class distribution. Future studies should also periodically retrieve new vulnerabilities from the CVE Details website in order to update the database, given that this process is automated. Finally, the file and class datasets could also be subject to a similar kind of analysis.

References

[1] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350. IEEE, 2006.

[2] Areej Algaith, Paulo Nunes, Fonseca Jose, Ilir Gashi, and Marco Vieira. Finding sql injection and cross site scripting vulnerabilities with diverse static analysis tools. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 57–64. IEEE, 2018.

[3] Omar H Alhazmi and Yashwant K Malaiya. Modeling the vulnerability discovery process. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10–pp. IEEE, 2005.

[4] Omar H Alhazmi and Yashwant K Malaiya. Quantitative vulnerability assessment of systems software. In *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pages 615–620. IEEE, 2005.

[5] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Software metrics and security vulnerabilities: dataset and exploratory study. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 37–44. IEEE, 2016.

[6] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156. IEEE, 2016.

[7] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. Schemaspy analysis of software - all relationships. `https://eden.dei.uc.pt/~nmsa/metrics-dataset/relationships.html`, 2016. Accessed: 2020-10-29.

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

[9] Aileen G Bacudio, Xiaohong Yuan, Bei-Tseng Bill Chu, and Monique Jones. An overview of penetration testing. *International Journal of Network Security & Its Applications*, 3(6):19, 2011.

[10] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[11] João R. Campos, Marco Vieira, and Ernesto Costa. Propheticus: Machine learning framework for the development of predictive models for reliable and secure software. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 173.–182, 2019.

[12] Boris Chernis and Rakesh Verma. Machine learning methods for software vulnerability detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, pages 31–39, 2018.

[13] Brian Chess and Jacob West. *Secure programming with static analysis*, pages 3–4, 21–24. Pearson Education, 2007.

[14] Shyam R Chidamber and Chris F Kemerer. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, 1991.

[15] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[16] Istehad Chowdhury and Mohammad Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1963–1969, 2010.

[17] IEEE Computer Society. Standards Coordinating Committee, Institute of Electrical, Electronics Engineers, and IEEE Standards Board. *IEEE Standard Glossary of Software Engineering Terminology*, page 31. IEEE Std. IEEE, 1990. ISBN 9781559370677.

[18] The MITRE Corporation. Cve mitre - terminology. `https://cve.mitre.org/about/terminology.html`, 10 2020. Accessed: 2020-11-21.

[19] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*, pages 18–24. Pearson Education, 2006.

[20] Michael Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

[21] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

[22] The Open Web Application Security Project Foundation. Owasp top 10 - 2017 - the ten most critical web application security risks. `https://owasp.org/www-project-top-ten/2017/`, 2017. Accessed: 2020-11-21.

[23] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications*, pages 581–591. Springer, 2015.

[24] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview. *ArXiv*, abs/2008.05756, 2020.

[25] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[26] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE transactions on Software Engineering*, 7(5):510–518, 1981.

[27] Willy Jimenez, Amel Mammar, and Ana Cavalli. Software vulnerabilities, prevention and detection methods: A review1. *Security in model-driven architecture*, 215995: 215995, 2009.

[28] SciTools Kevin Groke. What metrics does understand have? `https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-`, 2021. Accessed: 2021-10-25.

[29] Kaspersky Lab. Measuring financial impact of it security on businesses - it security risks report 2016. Technical report, Kaspersky Lab, 2016.

[30] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[31] Peng Li and Baojiang Cui. A comparative study on software vulnerability static analysis techniques and tools. In *2010 IEEE international conference on information theory and information security*, pages 521–524. IEEE, 2010.

[32] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: A survey. In *2012 fourth international conference on multimedia information networking and security*, pages 152–156. IEEE, 2012.

[33] Fabio Massacci and Viet Hung Nguyen. An empirical methodology to evaluate vulnerability discovery models. *IEEE Transactions on Software Engineering*, 40(12): 1147–1162, 2014.

[34] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320, 1976.

[35] Nádia Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. Software metrics as indicators of security vulnerabilities. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227. IEEE, 2017.

[36] Satwik Mishra. Handling imbalanced data: Smote vs. random undersampling. *International Research Journal of Engineering and Technology (IRJET)*, 4(8), 2017.

[37] Nuno Neves, Joao Antunes, Miguel Correia, Paulo Verissimo, and Rui Neves. Using attack injection to discover new vulnerabilities. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 457–466. IEEE, 2006.

[38] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.

[39] National Institute of Standards and Technology. National vulnerability database - vulnerabilities. `https://nvd.nist.gov/vuln`, 2020. Accessed: 2020-11-21.

[40] José D'Abruzzo Pereira and Marco Vieira. On the use of open-source c/c++ static analysis tools in large projects. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 97–102. IEEE, 2020.

[41] José D'Abruzzo Pereira, João R. Campos, and Marco Vieira. Machine learning to combine static analysis alerts with software metrics to detect security vulnerabilities: An empirical study. In *2021 17th European Dependable Computing Conference (EDCC)*, 2021.

[42] José D'Abruzzo Pereira, **João Henggeler Antunes**, and Marco Vieira. On building a vulnerability dataset with static information from the source code. In *Safety, Security, and Privacy in Complex Artificial Intelligence based Systems (SAFELIFE 2021)*, 2021. (accepted).

[43] Andrey Petukhov and Dmitry Kozlov. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. *Computing Systems Lab, Department of Computer Science, Moscow State University*, pages 1–120, 2008.

[44] IBM Security. Cost of a data breach report 2020. Technical report, Ponemon Institute, 2020.

[45] Miltiadis Siavvas, Dionysios Kehagias, and Dimitrios Tzovaras. A preliminary study on the relationship among software metrics and specific vulnerability types. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 916–921. IEEE, 2017.

[46] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering*, 29(4):297–310, 2003.

[47] Kazi Zakia Sultana. Towards a software vulnerability prediction model using traceable code patterns and software metrics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1022–1025. IEEE, 2017.

[48] Rahul Telang and Sunil Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software engineering*, 33(8):544–557, 2007.

[49] Bogdan Vasilescu, Alexander Serebrenik, and Mark Van den Brand. By no means: A study on aggregating software metrics. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, pages 23–26, 2011.

[50] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*, pages 149–151, 171–173. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2nd edition, 2005.

[51] R. Zhang, S. Huang, Z. Qi, and H. Guan. Combining static and dynamic analysis to discover software vulnerabilities. In *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 175–181, 2011.

[52] Serkan Özkan. Current cvss score distribution for all vulnerabilities. `https://www.cvedetails.com/cvss-score-distribution.php`, 2020. Accessed: 2021-01-02.

Appendices

This page is intentionally left blank.

Appendix A - Software Metrics Summary

Table 1 lists every software metric used in the created datasets. While most metrics were generated using Understand[1] version 4.0.837, some had to be aggregated in our own scripts using the sum, mean, and maximum functions. Additionally, some metrics only exist in specific datasets depending on the kind of code unit. These distinctions, along with a description adapted from the Understand tool's website [28], are also included in this summary table.

[1] https://www.scitools.com/

Name	File Metric	Function Metric	Class Metric	Description
AltAvgLineBlank	X		X	Average number of blank lines for all nested functions or methods, including inactive regions.
AltAvgLineCode	X		X	Average number of lines containing source code for all nested functions or methods, including inactive regions.
AltAvgLineComment	X		X	Average number of lines containing comment for all nested functions or methods, including inactive regions.
AltCountLineBlank	X	X	X	Number of blank lines, including inactive regions.
AltCountLineCode	X	X	X	Number of lines containing source code, including inactive regions.
AltCountLineComment	X	X	X	Number of lines containing comment, including inactive regions.
AvgCountInput	X			Average Fan-In (FANIN). Aggregated by our scripts.
AvgCountOutput	X			Average Fan-Out (FANOUT). Aggregated by our scripts.
AvgCyclomatic	X		X	Average cyclomatic complexity for all nested functions or methods.
AvgCyclomaticModified	X		X	Average modified cyclomatic complexity for all nested functions or methods.
AvgCyclomaticStrict	X		X	Average strict cyclomatic complexity for all nested functions or methods.
AvgEssential	X		X	Average Essential complexity for all nested functions or methods.
AvgLine	X		X	Average number of lines for all nested functions or methods.
AvgLineBlank	X		X	Average number of blank for all nested functions or methods.
AvgLineCode	X		X	Average number of lines containing source code for all nested functions or methods.
AvgLineComment	X		X	Average number of lines containing comment for all nested functions or methods.
AvgMaxNesting	X			Average maximum nesting level of control constructs. Aggregated by our scripts.
CountClassBase			X	Number of immediate base classes.
CountClassCoupled			X	Number of other classes coupled to. Also known as Coupling Between Objects (CBO).
CountClassDerived			X	Number of immediate subclasses. Also known as Number of Children (NOC).
CountDeclClass	X			Number of classes.
CountDeclClassMethod			X	Number of class methods.
CountDeclClassVariable			X	Number of class variables.
CountDeclFunction	X			Number of functions.
CountDeclInstanceMethod			X	Number of instance methods.
CountDeclInstanceVariable			X	Number of instance variables.
CountDeclInstanceVariablePrivate			X	Number of private instance variables.
CountDeclInstanceVariableProtected			X	Number of protected instance variables.
CountDeclInstanceVariablePublic			X	Number of public instance variables.
CountDeclMethod			X	Number of local methods.
CountDeclMethodAll			X	Number of methods, including inherited ones. Also known as Response for a Class (RFC).
CountDeclMethodConst			X	Number of local const methods.
CountDeclMethodFriend			X	Number of local friend methods.
CountDeclMethodPrivate			X	Number of local private methods.
CountDeclMethodProtected			X	Number of local protected methods.
CountDeclMethodPublic			X	Number of local public methods.
CountInput		X		Number of calling subprograms plus global variables read. Also known as FANIN.
CountLine	X	X	X	Number of all lines. Also known as Lines of Code (LOC).
CountLineBlank	X	X	X	Number of blank lines.
CountLineCode	X	X	X	Number of lines containing source code. Also known as Source Lines of Code (SLOC).
CountLineCodeDecl	X	X	X	Number of lines containing declarative source code.
CountLineCodeExe	X	X	X	Number of lines containing executable source code.
CountLineComment	X	X	X	Number of lines containing comment.
CountLineInactive	X	X	X	Number of inactive lines.
CountLinePreprocessor	X	X	X	Number of preprocessor lines.
CountOutput		X		Number of called subprograms plus global variables set. Also known as FANOUT.
CountPath		X		Number of possible paths, not counting abnormal exits or gotos.
CountSemicolon	X	X		Number of semicolons.
CountStmt	X	X	X	Number of statements.
CountStmtDecl	X	X	X	Number of declarative statements.
CountStmtEmpty	X	X	X	Number of empty statements.
CountStmtExe	X	X	X	Number of executable statements.
Cyclomatic		X		Cyclomatic complexity.
CyclomaticModified		X		Modified cyclomatic complexity.
CyclomaticStrict		X		Strict cyclomatic complexity.
Essential		X		Essential complexity.
HenryKafura	X			Sum of the number of combinations from an input source to an output destination in a function. Also known as the Henry Kafura Size (HK). Aggregated by our scripts using the metrics CountInput, CountOutput, and CountLineCodeExe.
Knots		X		Measure of overlapping jumps.
MaxCountInput	X			Maximum FANIN. Aggregated by our scripts.
MaxCountOutput	X			Maximum FANOUT. Aggregated by our scripts.
MaxCyclomatic	X		X	Maximum cyclomatic complexity of all nested functions or methods.
MaxCyclomaticModified	X		X	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	X		X	Maximum strict cyclomatic complexity of nested functions or methods.
MaxEssential	X		X	Maximum essential complexity of all nested functions or methods.
MaxEssentialKnots		X		Maximum Knots after structured programming constructs have been removed.
MaxInheritanceTree			X	Maximum Depth of Inheritance Tree (DIT).
MaxMaxNesting	X			Maximum maximum nesting level of control constructs. Aggregated by our scripts.
MaxNesting	X	X	X	Maximum nesting level of control constructs.
MinEssentialKnots		X		Minimum Knots after structured programming constructs have been removed.
PercentLackOfCohesion			X	100% minus the average cohesion for package entities. Also known as Lack of Cohesion in Methods (LCOM).
RatioCommentToCode	X	X	X	Ratio of comment lines to code lines.
SumCountClassBase	X			Sum of the number of immediate base classes. Aggregated by our scripts.
SumCountClassCoupled	X			Sum of the CBO. Aggregated by our scripts.
SumCountClassDerived	X			Sum of the NOC. Aggregated by our scripts.
SumCountDeclMethodAll	X			Sum of the RFC. Aggregated by our scripts.
SumCountInput	X			Sum of the FANIN. Aggregated by our scripts.
SumCountOutput	X			Sum of the FANOUT. Aggregated by our scripts.
SumCountPath	X			Sum of the number of possible paths, not counting abnormal exits or gotos. Aggregated by our scripts.
SumCyclomatic	X		X	Sum of cyclomatic complexity of all nested functions or methods. Also known as Weighted Methods Per Class (WMC).
SumCyclomaticModified	X		X	Sum of modified cyclomatic complexity of all nested functions or methods.
SumCyclomaticStrict	X		X	Sum of strict cyclomatic complexity of all nested functions or methods.
SumEssential	X		X	Sum of essential complexity of all nested functions or methods.
SumMaxInheritanceTree	X			Sum of the maximum DIT. Aggregated by our scripts.
SumMaxNesting	X			Sum of the maximum nesting level of control constructs. Aggregated by our scripts.
SumPercentLackOfCohesion	X			Sum of the LCOM. Aggregated by our scripts.

Table 1: A summary of the software metrics present in the code unit datasets. This includes metrics generated by the Understand tool [28] version 4.0.837 and any new ones aggregated using our scripts.

This page is intentionally left blank.

Appendix B - Code and Configuration Files Documentation

The following list describes the purpose of each Python script developed during the course of our work. These include scraping information from online platforms, interfacing with each C/C++ project's version control system, inserting and updating rows in the MySQL vulnerability database, and running any third-party tools such as Static Analysis Tools (SATs), Understand, or Propheticus.

All scripts and any other files used throughout development can be found in this project's GitHub repository[2]. Each item below represents the path of a script relative to the "scripts" directory inside this repository. Though every script uses underscore characters to separate words in its filename, we may sometimes replace them with hyphens here for ease of formatting in LaTeX.

These were developed using Python version 3.8 while making use of the third-party modules listed in the "scripts/requirements.txt" file. The scripts were executed in machines running Windows 10 and Ubuntu 20.04. Because various scripts depend on the output of previous ones, the list below presents each item in the order of execution. Other miscellaneous scripts or general-purpose modules appear at the end.

- **collect_vulnerabilities.py**: collects any vulnerabilities associated with the C/C++ projects by scraping the CVE Details website. The collected vulnerability data includes the Common Vulnerabilities and Exposure (CVE) identifier, publish date, Common Vulnerability Scoring System (CVSS) score, how it impacts different security properties, vulnerability types, Common Weakness Enumeration (CWE) identifier, and any Uniform Resource Locators (URLs) that link to relevant websites such as a project's bug tracker or security advisory platforms. For each project, this information is saved to a Comma-Separated Values (CSV) file;

- **find_affected_files.py**: finds any files affected by vulnerabilities associated with the C/C++ projects by querying their version control systems. This information includes the file's path, a list of CVEs, the Git commit hash where the vulnerability was patched (neutral file), the commit hash immediately before (vulnerable file), a list of line ranges where the file modifications occurred, and a list of functions and classes present in the file. Any data regarding functions and classes is collected using the Clang compiler[3] via a third-party Python module. This script uses the CSV files generated by **collect_vulnerabilities.py** to create its own CSVs. On a Windows machine, the correct Python version (32 or 64-bit) must be used so that it matches Clang's Dynamic Link Library (DLL);

- **create_file_timeline.py**: creates a timeline of files starting at each project's first commit and going through every commit that was affected by a vulnerability. The script only retrieves information about files that were actually modified in each commit (and not every file in the repository), reducing the time it takes to execute. This information includes the file's path, vulnerability status, the commit's hash, author date and tag name, a list of line ranges where the file modifications occurred, and a list of functions and classes present in the file. This script uses the CSV files generated by **find_affected_files.py** to create its own CSVs;

- **fix-neutral-code-unit-status-in-affected-files-and-file-timeline.py**:

[2] https://github.com/joaohenggeler/uc-masters-software-vulnerabilities
[3] https://clang.llvm.org/

updates the CSV files generated after running **find-affected-files.py** and **create-file-timeline.py** by fixing the vulnerability status of any neutral code units. This was done to fix a mistake introduced by a bug in the previous two scripts without having to run them again;

- **alter_engines_in_database.py**: converts the engine from any table using MyISAM to InnoDB in the vulnerability database. The Structured Query Language (SQL) scripts used to import the original dataset developed by Alves et al. [5] specify MyISAM as the default engine for every table. This change is advantageous since this type of engine does not support foreign keys or transactions[4], while InnoDB does;

- **import_extra_time_files_functions_classes_in_database.py**: imports the data from the original EXTRA_TIME_FILES, EXTRA_TIME_FUNCTIONS, and EXTRA_TIME_CLASS tables into the database. This operation is provided separately as it can take a long time to complete;

- **merge_files_functions_classes_in_database.py**: merges the FILES_*, FUNCTIONS_*, and CLASSES_* tables into a single one for each project and code unit type. For example, the original tables for any files belonging to the Xen project are kept in the tables FILES_3_ARCH, FILES_3_TOOLS, and FILES_3_XEN, where the value 3 is the project's identifier. After running this script, a new table called FILES_3 containing all this data is created;

- **collect_missing_cwes.py**: collects any missing CWE values associated with the vulnerabilities in the C/C++ projects by scraping the CVE Details website. This is done to enrich the vulnerabilities in the original dataset with a specific category. For each project, this information is saved to a CSV file;

- **alter_vulnerabilities_in_database.py**: makes the following structural changes to the VULNERABILITIES table in the database: adds columns to represent each vulnerability's CWE and project identifier; sets the values of this last identifier and adds a foreign key relationship that references the projects in the REPOSITORIES_SAMPLE table; creates a new numeric primary key column that increments itself automatically; applies this last process to the PATCHES table to preserve the foreign key relationship with the VULNERABILITIES table;

- **insert_vulnerabilities_in_database**: inserts the data from any CSV files generated by **collect_vulnerabilities.py** into the VULNERABILITIES table into the database. Before running this script, this table must be first modified using **alter_vulnerabilities_in_database.py**;

- **update_missing_cwes_in_database.py**: updates any missing CWE values associated with the vulnerabilities in the C/C++ projects. Before running this script, this data must be first collected using **collect_missing_cwes.py**;

- **insert_patches_in_database**: inserts the data from any CSV files generated by **create_file_timeline.py** into the PATCHES table in the database. Before running this script, the previously collected vulnerabilities must be first inserted using **insert_vulnerabilities_in_database**;

- **generate_metrics.py**: generates the software metrics for any files affected by vulnerabilities associated with the C/C++ projects using the Understand tool. This

[4] `https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html`

information includes the file's path, whether it was vulnerable or not, the associated commit hash where this specific file version originated from, and various different software metrics at a file, function, and class level. The metrics generated by this script are specified in Appendix 6. This script uses the CSV files generated by **create_file_timeline.py** to create its own CSVs;

- **verify_output_metrics.py**: verifies the output of **generate_metrics.py** by checking if each commit has at least one CSV file associated with it. This was done to find a list of commit hashes that were missing their respective software metrics due to an occasional runtime error when executing the Understand tool;

- **split_and_update_metrics**: splits the software metrics generated by **generate_metrics.py** according to their code units types (files, functions, or classes), and computes new ones by aggregating them. The following file-level metrics are created using the function and class-level ones: SumCountPath, MaxCountInput, AvgCountInput, SumCountInput, MaxCountOutput, AvgCountOutput, SumCountOutput, MaxMaxNesting, AvgMaxNesting, SumMaxNesting, and HenryKafura. This script outputs three CSV files (file, function, and class metrics) for each CSV generated by **generate_metrics.py**;

- **compare_metrics_in_database.py**: compares the file-level metrics generated by **split_and_update_metrics** with the ones previously inserted in the database. This was done to determine if Understand 6.0.1046 computed software metrics the same way as version 4.0.837, which was used by Alves et al. [5] when creating the original dataset;

- **alter-functions-and-classes-in-database.py**: adds two columns (BeginLine and EndLine) to the FUNCTIONS_* and CLASSES_* tables in the database, ensuring that the line ranges these code units occupy can be stored. Before running this script, the code unit tables must be merged using **merge-files-functions-classes-in-database.py**;

- **insert_metrics_in_database.py**: inserts the data from any CSV files generated by **split_and_update_metrics.py** into the FILES_*, FUNCTIONS_*, CLASSES_*, and EXTRA_TIME_* tables. Before running this script, the following scripts must be first executed: **insert-patches-in-database.py**, **alter-functions-and-classes-in-database.py**, and (previously) **alter-int-primary-keys-to-big-int-in-database.py**;

- **aggregate-ck-file-metrics-in-database.py**: updates six software metric columns in the FILES_* tables whose value was not previously aggregated by **split-and-update-metrics** due to an oversight during development. The file-level metrics in question are the aggregates of the following class-level metrics: DIT, NOC, RFC, CBO, LCOM, and CountClassBase. These are computed by applying the sum, mean, and maximum functions to the corresponding rows in the CLASSES_* table. Before running this script, the previously generated metrics must be first inserted using **insert_metrics_in_database.py**;

- **create_alert_and_cwe_tables_in_database.py**: creates any tables which are necessary to insert security alerts and additional information about CWEs in the database. These include the following tables: VULNERABILITY_CATEGORY, CWE_INFO, SAT, RULE, RULE_CWE_INFO, ALERT, ALERT_FUNCTION, and ALERT_CLASS. The tables VULNERABILITY_CATEGORY, CWE_INFO,

and SAT are populated with values specified in the static configuration file. Before running this script, the code unit tables must be merged using **merge-files-functions-classes-in-database.py**;

- **generate_alerts.py**: generates the security alerts for any files affected by vulnerabilities associated with the C/C++ projects using specific SATs. This information includes the file's path, whether it was vulnerable or not, the associated commit hash where this specific file version originated from, and various different security alerts. This script uses the CSV files generated by **create_file_timeline.py** to create its own CSVs. Although this script was supposed to be run alongside **generate_metrics.py**, the security alerts used for this work were generated by one of the advisors using their own scripts. This script is currently only configured to run the Cppcheck SAT but may be easily expanded to include others;

- **list_neutral_commits.py**: lists any neutral commit hashes affected by vulnerabilities associated with the C/C++ projects. This simple script was used to specify which commits should be processed by the SATs in order to generate security alerts. This script uses the CSV files generated by **find_affected_files.py** to create its own CSVs;

- **create_indexes_in_database.py**: adds two indexes to the PATCHES and ALERT tables in the database in order to improve the performance when inserting alerts and later generating the raw dataset. Before running this script, the security alerts table must be created using **create_alert_and_cwe_tables_in_database.py**;

- **insert_alerts_in_database.py**: inserts the alert data from any CSV files present in the directory specified in the dynamic configuration file into the RULE, CWE_INFO, RULE_CWE_INFO, ALERT, ALERT_FUNCTION, and ALERT_CLASS tables in the database. Before running this script, the following scripts must be first executed:

  insert-metrics-in-database.py, **create-alert-and-cwe-tables-in-database.py**, and **create-indexes-in-database.py**;

- **build_raw_dataset_from_database.py**: exports a raw dataset composed of every project's code unit from the database. A CSV file is created for each code unit kind. Each dataset is composed of samples containing software metrics and security alert occurrences of different files, functions, or classes. Before running this script, the following scripts must be first executed: **insert-metrics-in-database.py**, **aggregate-ck-file-metrics-in-database.py**, and **insert-alerts-in-database.py**;

- **merge_raw_datasets**: merges any raw datasets generated by **build-raw-dataset-from-database.py** into a single one for each code unit kind. This script exists to allow the creation of a dataset containing samples from all projects without running into disk space exhaustion problems when exporting a large number of rows from the database. It is also used to remove a specific percentage of neutral samples, and to group any vulnerable multiclass labels that fall under a given threshold;

- **build_propheticus_dataset_from_raw_dataset.py**: converts any raw dataset merged by **merge_raw_datasets.py** to a specific version that can be parsed by the Propheticus tool. For each processed dataset, three files are created: one listing the total number of samples (.info.txt), one specifying the dataset's columns and their data types (.headers.txt), and another containing the data itself (.data.txt);

  The following two scripts are kept in "propheticus/instances/vulnerability-prediction-final" due to a requirement of the Propheticus tool:

- **InstanceConfig.py**: defines a class that overrides any default configurations used by Propheticus when processing the datasets generated by **build-propheticus-dataset-from-raw-dataset.py**;

- **InstanceBatchExecution.py**: defines a class that specifies how Propheticus should batch process the datasets generated by **build-propheticus-dataset-from-raw-dataset.py**. This includes choosing which class labels to consider and listing the dimensionality reduction techniques, data balancing methods, and classification algorithms to apply. After training and testing classifiers using every possible combination of these values, Propheticus will output various prediction results and performance metrics to disk. Must be run on the 64-bit version of Python since Propheticus depends on the Tensorflow module. An older version of the xlrd module (e.g. 1.2.0) is also required for some functionalities. On a Linux machine, the unixodbc package must be installed so that the pypyodbc Python module can be imported correctly by Propheticus;

- **export_propheticus_result_comparison.py**: exports the most relevant values from a given Propheticus results comparison Excel log to a CSV file. Also shows the best results for each label and performance metric. Before passing the Excel file to this script, it must be first generated by running the experiments in Propheticus and then telling it to compare all results;

- **validate_datasets_using_temporal_windows.py**: validates any datasets created by **merge_raw_datasets** by rerunning the best Propheticus configurations with a new data partitioning strategy: use a range of vulnerability years as the training subset, and the next year as the testing subset. For example, the following list of tuples where the first element represents the training range, and the second the testing year: (2008-2012, 2013), (2009-2013, 2014), and so on until (2014-2018, 2019). Here, a sliding window of five years is considered. Because we only had to use the algorithms and parameters that yielded the best results when run in the Propheticus tool, this script only implements the following machine learning techniques:

  - **Classification Algorithms**: Random Forest (RF), Bagging;
  - **Dimensionality Reduction**: variance threshold;
  - **Data Balancing**: random undersampling.

- **plot_temporal_windows_results.py**: plots the performance metrics generated after rerunning the best Propheticus configurations using temporal sliding windows with **validate-datasets-using-temporal-windows.py**. For each configuration, the following two files are created: 1) a text file containing part of a table with the performance metric values; 2) an image containing nine lines (for three metrics and three window sizes).

- **modules/common.py**: defines any general-purpose functionalities used by all scripts, including functions for logging information, loading configuration files, and serializing data;

- **modules/cve.py**: defines a class that represents a software vulnerability and that contains methods for scraping its data from the CVE Details website;

- **modules/database.py**: defines a class that represents a MySQL database connection and that contains methods for querying its information;

- **modules/project.py**: defines a class that represents a C/C++ project and that contains methods for interfacing with its vulnerabilities and source files. Multiple subclasses are defined given each project's requirements, such as scraping information from security advisories and its version control system in different ways;
- **modules/sats.py**: defines any classes that represent third-party tools used to perform Static Code Analysis (SCA) on a project's source files. This includes the Understand tool and the Cppcheck and Flawfinder SATs;
- **modules/scraping.py**: defines any methods and classes that are used to download and parse vulnerability metadata from websites;
- **split_csv.py**: splits a CSV file into multiple ones. This script was meant to bypass the 100 MB file limit in this work's Git repository. However, it was left unused since the resulting CSV files could be easily compressed due to their nature.

In the previously documented Python scripts, the terms static and dynamic configuration files are used. These refer to two files that store parameters that influence how most scripts behave. Their names are derived from the fact that some values are permanent (static) while others depend on the scripts' environment or desired outcome (dynamic). This information is kept in two files, **static_config.json** and **dynamic_config.json** in the **scripts/modules/config** directory, using the JavaScript Object Notation (JSON) format. The dynamic configuration file is kept private since it contains paths that depend on the machine it's located on, as well as sensitive data (e.g. the MySQL database credentials and GitHub personal access tokens). As such, a generic template file called **dynamic_config_template.json** was included in this work's repository in its place.

The contents of the static configuration file and an adaptation of the dynamic template one can be found in Listings 1 and 2, respectively. What follows is a description of each parameter stored in the static configuration file.

- **projects**: maps each project's full name to a JSON object that contains the following parameters. The class defined in **modules/project.py** stores these values as attributes.
  - **short_name**: a shorter and lowercase version of the project's full name. Used to name the resulting CSV files created by the scripts;
  - **database_id**: the primary key value that identifies the project in the database. Known as the R_ID column in the various tables;
  - **database_name**: the suffix that identifies each project's views in the database. Also used as the name of the subdirectory that contains each project's SQL import scripts;
  - **github_data_name**: the name of the directory in an external repository that stores this project's security alerts;
  - **vendor_id** and **product_id**: two values that together uniquely identify a project in CVE Details. Used to list and scrape vulnerabilities from this website. The value of **product_id** may be null if all software from the vendor is to be considered;
  - **url_pattern**: a regular expression used to remove any URLs unrelated to the project when scraping the references section from each vulnerability's page in the CVE Details website;

  - **master_branch**: the name of the main branch in the project's repository. Used to filter commit hashes if the option **scrape_all_branches** is disabled;
  - **language**: the project's programming language. Only "c" and "c++" are supported;
  - **include_directory_path**: the path to the project's header files, relative to the repository's root directory. May be null if no such directory exists.

- **sats**: maps each SAT's full name to an object that contains the following parameters. The classes defined in **modules/sats.py** store these values as attributes. These are set to null for Understand since this tool is not used to generate alerts, even though it performs SCA.
  - **database_name**: the SAT's name as shown in the SAT_NAME column of the SAT table in the database;
  - **github_data_name**: the name of the subdirectory in an external repository that stores the security alerts generated by this SAT.

- **vulnerability_categories**: maps a vulnerability category to a list of CWEs that belong to it, as is also defined in Table 3.3. These values are inserted in the VULNERABILITY_CATEGORY and CWE_INFO tables in the database when the alert tables are first created. This object is also used to assign each class label a numeric value (the category's index) when building the code unit datasets;

- **target_labels**: a list of every target label considered in the datasets. Used to stop their respective columns from being interpreted as features when validating the dataset;

- **http_headers**: a collection of Hypertext Transfer Protocol (HTTP) headers belonging to different web browsers. These are used by the ScrapingManager class defined in **modules/scraping.py** to download web pages without having the requests be rejected by the server.

```
{
  "projects":
  {
    "Glibc":
    {
      "short_name": "glibc",
      "database_id": 5,
      "database_name": "glibc",
      "github_data_name": "glibc",
      "vendor_id": 72,
      "product_id": 767,
      "url_pattern": "sourceware",
      "master_branch": "master",
      "language": "c",
      "include_directory_path": "include"
    },

    "Apache HTTP Server":
    {
      "short_name": "apache",
      "database_id": 4,
      "database_name": "httpd",
      "github_data_name": "httpd",
      "vendor_id": 45,
```

```
      "product_id": 66,
      "url_pattern": "apache",
      "master_branch": "trunk",
      "language": "c",
      "include_directory_path": "include"
    },

    "Xen":
    {
      "short_name": "xen",
      "database_id": 3,
      "database_name": "xen",
      "github_data_name": "xen",
      "vendor_id": 6276,
      "product_id": null,
      "url_pattern": "xen",
      "master_branch": "master",
      "language": "c",
      "include_directory_path": "xen/include"
    },

    "Linux Kernel":
    {
      "short_name": "kernel",
      "database_id": 2,
      "database_name": "kernel",
      "github_data_name": "linux",
      "vendor_id": 33,
      "product_id": 47,
      "url_pattern": "linux|kernel|redhat",
      "master_branch": "master",
      "language": "c",
      "include_directory_path": "include"
    },

    "Mozilla":
    {
      "short_name": "mozilla",
      "database_id": 1,
      "database_name": "mozilla",
      "github_data_name": "mozilla",
      "vendor_id": 452,
      "product_id": null,
      "url_pattern": "mozilla",
      "master_branch": "master",
      "language": "c++",
      "include_directory_path": null
    }
  },

  "sats":
  {
    "Understand":
    {
      "database_name": null,
      "github_data_name": null
    },

    "Cppcheck":
    {
      "database_name": "Cppcheck",
      "github_data_name": "cppcheck"
    },
```

```
    "Flawfinder":
    {
      "database_name": "Flawfinder",
      "github_data_name": "flawfinder"
    }
  },

  "vulnerability_categories":
  {
    "Memory Management":      [119, 362, 399, 416, 476, 824],
    "Input Validation":       [20, 78, 79, 91, 94, 134, 189],
    "Permission":           [255, 264, 269, 284, 287, 352],
    "Data Protection":        [199, 200],
    "Coding Practices":       [17, 19, 254],
    "Cryptography":         [310],
    "System Configuration":     [16],
    "File Management":        [22, 59],

    "Output Encoding":        [],
    "Error Handling and Logging":   [],
    "Communication Security":     [],
    "Database Security":      []
  },

    "target_labels": ["binary_label", "grouped_multiclass_label"],

  "http_headers":
  {
    "Chrome":
    {
      "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
    image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;
    v=b3;q=0.9",
      "Accept-Encoding": "identity",
      "Accept-Language": "en-US,en;q=0.9",
      "Sec-Fetch-Dest": "document",
      "Sec-Fetch-Mode": "navigate",
      "Sec-Fetch-Site": "none",
      "Sec-Fetch-User": "?1",
      "Upgrade-Insecure-Requests": "1",
      "User-Agent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit
    /537.36 (KHTML, like Gecko) Chrome/86.0.4240.75 Safari/537.36"
    },

    "Firefox":
    {
      "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
    image/webp,*/*;q=0.8",
      "Accept-Encoding": "identity",
      "Accept-Language": "en-US,en;q=0.5",
      "Dnt": "1",
      "Referer": "https://duckduckgo.com/?q=test&t=h_&ia=web",
      "Sec-Gpc": "1",
      "Upgrade-Insecure-Requests": "1",
      "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:86.0)
    Gecko/20100101 Firefox/86.0"
    },

    "Edge":
    {
      "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
    image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9"
```

```
    ,
      "Accept-Encoding": "identity",
      "Accept-Language": "en-US,en;q=0.9",
      "Sec-Fetch-Dest": "document",
      "Sec-Fetch-Mode": "navigate",
      "Sec-Fetch-Site": "none",
      "Sec-Fetch-User": "?1",
      "Upgrade-Insecure-Requests": "1",
      "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit
    /537.36 (KHTML, like Gecko) Chrome/88.0.4324.182 Safari/537.36 Edg
    /88.0.705.74"
    },

    "Internet Explorer":
    {
      "Accept": "text/html, application/xhtml+xml, image/jxr, */*",
      "Accept-Encoding": "identity",
      "Accept-Language": "en-US",
      "User-Agent": "Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv
    :11.0) like Gecko"
    }
  }
}
```

Listing 1: The static configuration JSON file used by the Python scripts.

The parameters of the dynamic configuration file are documented below.

- **debug**: a collection of debug options used to validate certain scripts before executing them to generate the final results;
  - **enabled**: whether the following debug parameters are used;
  - **min_hub_pages**: the minimum number of hub pages in CVE Details for a given project that have to exist to consider the following option. These "hub pages" list and link to a set number of vulnerabilities using a pagination process;
  - **hub_page_step**: defines a step size that skips some hub pages that were collected from CVE Details, provided there were enough of them. For example, a step of three would only consider the first, fourth, seventh, etc, pages. Used to test the vulnerability collection script with a smaller sample size;
  - **max_cves_per_hub_page**: the maximum number of vulnerabilities to collect from each hub page;
  - **use_random_sampling**: whether to draw a random sample of vulnerabilities from each hub page or to only consider the first ones. This is done using the sample size defined above;
  - **verify_different_unit_names**: whether to apply a verification step to the software metrics generation process using the parameter below;
  - **different_unit_names_ratio_limit**: defines a threshold that, if crossed during the metrics generation process, logs a debug message to the script's log file. The limit represents how close two code unit (function or class) names have to be before issuing a warning with both units' properties (the higher the computed ratio, the more similar the names). This is done because Understand only shows a code unit's name (and not its line range), meaning we have to associate them with the names in our previously created timeline CSV. During development, there was some concern that the script could be mishandling names that were very similar, but failed a strict equality condition.

- **recursion_limit**: sets the maximum number of recursive calls allowed by a Python function before halting a script. This is used to bypass this restriction in third-partly modules, such as Clang;
- **allowed_projects**: an object that specifies whether a given project should be considered when running a script;
- **output_directory_path**: the path to the directory where any input and output CSV files should be read or written to. May be relative to the script's working directory;
- **start_at_cve_hub_page**: which hub page to start from when collecting vulnerabilities from CVE Details. Used to continue scraping information if the script had to be stopped for whatever reason. May be set to null to start from the beginning;
- **scrape_all_branches**: whether to consider all commit hashes (true) or only the ones from the master branch of the project's repository (false) when collecting vulnerabilities. This was set to false throughout this work's development;
- **affected_files_csv_write_frequency**: how often to update the CSV files to disk when running **find_affected_files.py** and **create_file_timeline.py**. Measured in the number of commits;
- **neutral_after_author_date** and **neutral_before_author_date**: two parameters that specify a commit's author date range. Used to only list source files that were modified between these dates;
- **start_at_timeline_index**: which commit index to start from when creating the file timeline. Used to resume the execution of the script if it had to be stopped for whatever reason. May be set to null to start from the beginning;
- **start_at_checkout_commit_index**: which commit index to start from when performing the Git checkout operation in order to generate software metrics and security alerts. Used to resume the execution of the script if it had to be stopped for whatever reason. May be set to null to start from the beginning;
- **checkout_commit_index_list**: a list of commit indexes that should only be processed when performing the same operation as mentioned above. Used to retry the execution of Understand or SATs if they failed to generate results for specific commits, but succeeded for others. May be set to null to skip this commit filtering process;
- **clang_lib_path**: the path to the directory containing Clang's DLL (Windows) or shared library (Linux) file. May also be the path to the file itself;
- **dataset_path**: the path to the directory containing the original dataset and its SQL import scripts, as provided by Alves et al. [5];
- **extra_time_tables_to_import**: a list of suffixes for the EXTRA_TIME_* tables to import into the database;
- **data_repository_path**: the path to the directory containing the files from the security alert CSV repository mentioned above;
- **allowed_code_units**: an object that specifies whether a given code unit (file, function, or class) should be considered when running a script;

- **allowed_sats**: an object that specifies whether a given SAT (excluding Understand) should be considered when running a script;
- **dataset_neutral_sample_removal_ratio**: defines a percentage of neutral samples to remove from the dataset. Used as a preliminary data balancing technique when merging the raw datasets;
- **dataset_vulnerable_label_threshold**: defines a percentage of vulnerable samples for when to change each vulnerability category label to the grouped multiclass label. For example, if set to 0.05, then any vulnerability category that falls under 5% of the total number of vulnerable samples is converted to this new label. Used when merging the raw datasets;
- **dataset_filter_samples_ineligible_for_alerts**: whether to remove any function or class samples that could never be mapped to security alerts due to missing line numbers. Used when building the raw datasets;
- **dataset_filter_commits_without_alerts**: whether to remove samples associated with a commit without any inserted alerts. Used when building the raw datasets;
- **account_username** and **account_password**: the credentials of a user with root privileges. Used very sparingly for situations such as changing the file permissions and ownership of the raw dataset exported by the MySQL Daemon. Only used in a Linux environment;
- **propheticus**: a collection of parameters that are passed to the Propheticus tool when creating classifiers using the generated dataset. The total number of executed configurations is equal to the product of the number of labels, dimensionality reduction and data balancing techniques, and classification algorithm hyperparameters.
  - **max_thread_count**: the maximum number of threads to use when training and testing the models;
  - **labels**: which class labels from the dataset to consider as the target;
  - **seed_count**: the number of times each configuration is executed. The final performance metrics of a configuration are the average of this number;
  - **data_split**: an object containing the parameters for how the dataset should be partitioned. The **n_splits** option specifies the number of folds in the k-fold cross-validation technique;
  - **dimensionality_reduction**: a list of dimensionality reduction techniques as specified in the file **propheticus/configs/DimensionalityReduction.py** from the Propheticus source code;
  - **data_balancing**: a list of data balancing techniques as specified in the file **propheticus/configs/Sampling.py** from the Propheticus source code;
  - **classification_algorithms**: an object mapping each classification algorithm to a list of hyperparameters objects. This list must contain either null or JSON objects whose values are lists. If a hyperparameter list element is an object, then the cartesian product of these list values is used to build several configurations. Otherwise, if a list element is null, then a configuration with default hyperparameter values is assumed. These algorithms and their parameters are specified in the file **propheticus/configs/Classification.py** from the Propheticus source code.

- **temporal_window**: a collection of parameters that are passed to our own validation script when creating classifiers by partitioning the generated dataset according to each sample's vulnerability year (i.e. by using a temporal sliding window).
  - **num_runs**: the number of times each configuration is executed. The final performance metrics of a configuration are the average of this number;
  - **data_split**: an object containing the parameters for how the dataset should be partitioned. The **begin_test_year** option specifies the first year to consider as the testing subset. The previous years are considered as the training subset, where the number of years in this set is defined by the list of sizes in the **window_size** option. A size of null is interpreted as a variable-length window that includes every year before the testing one;
  - **default_algorithm_parameters**: an object mapping each classification algorithm to a set of parameters that should be assumed by default. Used to emulate the behavior of Propheticus where some classifiers have default parameters that are not explicitly set in each configuration;
  - **configurations**: a list of the best classifier configurations obtained after using the Propheticus tool to explore a wide array of parameter combinations. Note that, unlike the **classification_algorithms** option in the **propheticus** object, each configuration is executed as is, and no combination of parameters is performed;
- **projects**: maps each project's full name to an object that contains any dynamic parameters. Currently, the only supported option is **repository_path**, which points to the directory containing the project's repository;
- **sats**: maps each SAT's full name to an object that contains any dynamic parameters. Currently, the only supported option is **executable_path**, which points to the tool's executable binary. This value may be an absolute path, or the just the filename (provided this command is recognized by the operating system). It may also be null if a given tool is not installed;
- **database**: the credentials used to connect to the database, as well as any other additional options. All parameters defined in this object are passed as is to the MySQL connector class in Python[5]. For example, the option **sql_mode** may also be specified to change a session variable of the same name.
  - **host**: the hostname or address of the MySQL server;
  - **port**: the port of the MySQL server;
  - **user**: the username used to authenticate the connection;
  - **password**: the password used to authenticate the connection;
  - **database**: the name of the database where the queries will be executed.

```
{
  "debug":
  {
    "enabled": false,

    "min_hub_pages": 6,
    "hub_page_step": 5,
```

[5] https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html

```
    "max_cves_per_hub_page": 5,
    "use_random_sampling": true,

    "verify_different_unit_names": true,
    "different_unit_names_ratio_limit": 90
  },

  "recursion_limit": 1000000,

  "allowed_projects":
  {
    "glibc": false,
    "apache": false,
    "xen": true,
    "kernel": true,
    "mozilla": true
  },

  "output_directory_path": "output",

  "start_at_cve_hub_page": null,
  "scrape_all_branches": false,

  "affected_files_csv_write_frequency": 10,
  "neutral_after_author_date": "1990-01-01",
  "neutral_before_author_date": "2022-01-01",

  "start_at_timeline_index": null,

  "start_at_checkout_commit_index": null,
  "checkout_commit_index_list": null,

  "clang_lib_path": "/usr/lib/x86_64-linux-gnu/libclang-10.so.1",

  "dataset_path": "/home/admin/dataset",

  "extra_time_tables_to_import": ["files", "functions", "class"],

  "data_repository_path": "/home/admin/repositories/repository-name",

  "allowed_code_units":
  {
    "file": false,
    "function": true,
    "class": true
  },

  "allowed_sats":
  {
    "Cppcheck": false,
    "Flawfinder": true
  },

  "dataset_neutral_sample_removal_ratio": 0.80,
  "dataset_vulnerable_label_threshold": 0.05,
  "dataset_filter_samples_ineligible_for_alerts": true,
    "dataset_filter_commits_without_alerts": true,

    "account_username": "<Username>",
  "account_password": "<Password>",

  "propheticus":
  {
```

```
    "max_thread_count": 6,
    "labels": ["binary_label", "grouped_multiclass_label"],

    "seed_count": 5,
    "data_split": {"n_splits": 5},

    "dimensionality_reduction": [["variance"], ["variance", "correlation"
    ]],
    "data_balancing": [[], ["RandomUnderSampler"], ["RandomOverSampler"], [
    "RandomUnderSampler", "RandomOverSampler"]],

    "classification_algorithms":
    {
      "random_forests":
      [
        null,
        {
          "n_estimators": [100, 200],
          "criterion": ["gini"],
          "min_samples_split": [0.001, 2],
          "min_samples_leaf": [0.001, 1],
          "max_features": [null],
          "bootstrap": [true]
        }
      ],

      "bagging":
      [
        null,
        {
          "n_estimators": [100, 200],
          "max_features": [0.1, 0.55, 1.0],
          "bootstrap": [true]
        }
      ],

      "xgboost":
      [
        null,
        {
          "n_estimators": [100, 300],
          "learning_rate": [0.1, 0.3],
          "gamma": [0],
          "subsample": [1],
          "max_depth": [10, 30],
          "min_samples_split": [2, 5],
          "min_samples_leaf": [1, 4]
        }
      ]
    }
  },

  "temporal_window":
  {
    "num_runs": 30,
    "data_split": {"begin_test_year": 2013, "window_size": [null, 5, 10]},

    "default_algorithm_parameters":
    {
      "random_forests": {"n_jobs": -1},
      "bagging": {"n_jobs": -1}
    },

```

```
    "configurations":
    [
      {
        "name": "Best Precision (Binary) - Default", "target_label": "
    binary_label",
        "dimensionality_reduction": ["variance"], "data_balancing": ["
    RandomUnderSampler"],
        "classification_algorithm": "random_forests", "algorithm_parameters
    ": {"bootstrap": true, "criterion": "gini", "max_features": "auto", "
    min_samples_leaf": 1, "min_samples_split": 2, "n_estimators": 100}
      },

      {
        "name": "Best Recall (Binary)", "target_label": "binary_label",
        "dimensionality_reduction": ["variance"], "data_balancing": [],
        "classification_algorithm": "bagging", "algorithm_parameters": {"
    bootstrap": true, "max_features": 0.55, "n_estimators": 200}
      },

      {
        "name": "Best F-score (Binary) 1 of 2", "target_label": "
    binary_label",
        "dimensionality_reduction": ["variance"], "data_balancing": [],
        "classification_algorithm": "bagging", "algorithm_parameters": {"
    bootstrap": true, "max_features": 1.0, "n_estimators": 100}
      },

      {
        "name": "Best F-score (Binary) 2 of 2 - Default", "target_label": "
    binary_label",
        "dimensionality_reduction": ["variance"], "data_balancing": [],
        "classification_algorithm": "bagging", "algorithm_parameters": {"
    bootstrap": true, "max_features": 1.0, "n_estimators": 10}
      },

      {
        "name": "Best Precision (Multiclass) - Default", "target_label": "
    grouped_multiclass_label",
        "dimensionality_reduction": ["variance"], "data_balancing": ["
    RandomUnderSampler"],
        "classification_algorithm": "random_forests", "algorithm_parameters
    ": {"bootstrap": true, "criterion": "gini", "max_features": "auto", "
    min_samples_leaf": 1, "min_samples_split": 2, "n_estimators": 100}
      },

      {
        "name": "Best Recall (Multiclass) 1 of 2", "target_label": "
    grouped_multiclass_label",
        "dimensionality_reduction": ["variance"], "data_balancing": [],
        "classification_algorithm": "bagging", "algorithm_parameters": {"
    bootstrap": true, "max_features": 0.55, "n_estimators": 100}
      },

      {
        "name": "Best Recall (Multiclass) 2 of 2", "target_label": "
    grouped_multiclass_label",
        "dimensionality_reduction": ["variance"], "data_balancing": [],
        "classification_algorithm": "bagging", "algorithm_parameters": {"
    bootstrap": true, "max_features": 0.55, "n_estimators": 200}
      },

      {
        "name": "Best F-score (Multiclass) 1 of 2", "target_label": "
```

```
    grouped_multiclass_label",
178         "dimensionality_reduction": ["variance"],"data_balancing": [],
179         "classification_algorithm": "random_forests", "algorithm_parameters
    ": {"bootstrap": true, "criterion": "gini", "max_features": null, "
    min_samples_leaf": 1, "min_samples_split": 2, "n_estimators": 200}
180       },
181
182       {
183         "name": "Best F-score (Multiclass) 2 of 2", "target_label": "
    grouped_multiclass_label",
184         "dimensionality_reduction": ["variance"],"data_balancing": [],
185         "classification_algorithm": "bagging", "algorithm_parameters": {"
    bootstrap": true, "max_features": 1.0, "n_estimators": 200}
186       }
187     ]
188   },
189
190   "projects":
191   {
192     "Glibc":
193     {
194       "repository_path": "/home/admin/repositories/glibc"
195     },
196
197     "Apache HTTP Server":
198     {
199       "repository_path": "/home/admin/repositories/httpd"
200     },
201
202     "Xen":
203     {
204       "repository_path": "/home/admin/repositories/xen"
205     },
206
207     "Linux Kernel":
208     {
209       "repository_path": "/home/admin/repositories/linux"
210     },
211
212     "Mozilla":
213     {
214       "repository_path": "/home/admin/repositories/gecko-dev"
215     }
216   },
217
218   "sats":
219   {
220     "Understand":
221     {
222       "executable_path": "/home/admin/sats/understand/4.0.837/scitools/bin/
    linux64/und"
223     },
224
225     "Cppcheck":
226     {
227       "executable_path": "cppcheck"
228     },
229
230     "Flawfinder":
231     {
232       "executable_path": null
233     }
234   },
```

```
235
236   "database":
237   {
238     "host": "127.0.0.1",
239     "port": "3306",
240     "user": "<Username>",
241     "password": "<Password>",
242     "database": "software",
243     "sql_mode": "ONLY_FULL_GROUP_BY,NO_ZERO_IN_DATE,NO_ZERO_DATE,
      ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION"
244   }
245 }
```

Listing 2: An example of a possible dynamic configuration JSON file used by the Python scripts.

This page is intentionally left blank.

Appendix C - Results for the Exploratory Experiments

This Appendix contains confusion matrices showcasing how the best configurations presented in Section 4.2 predicted each function's vulnerability status. All configurations in this Appendix are specified in Table 4.5. The matrix associated with configuration C_5 is shown in Figure 4.2 of this previous section. The markers in each figure refer to the following grouped multiclass labels: Memory Management (MM), Neutral (N), Vulnerable With No Category (V(NC)), and Vulnerable With A Category (V(WC)). For a binary target label, only N and V(NC) are considered. All figures were generated by the Propheticus tool.

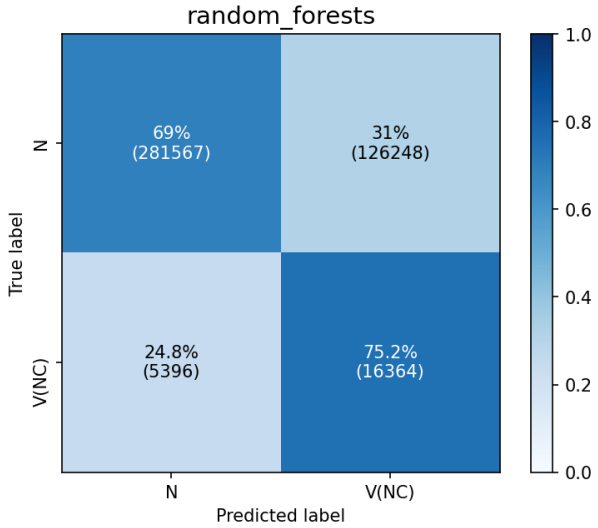


Figure 1: The confusion matrix showing the predictions of the classifier trained with configuration C_1.

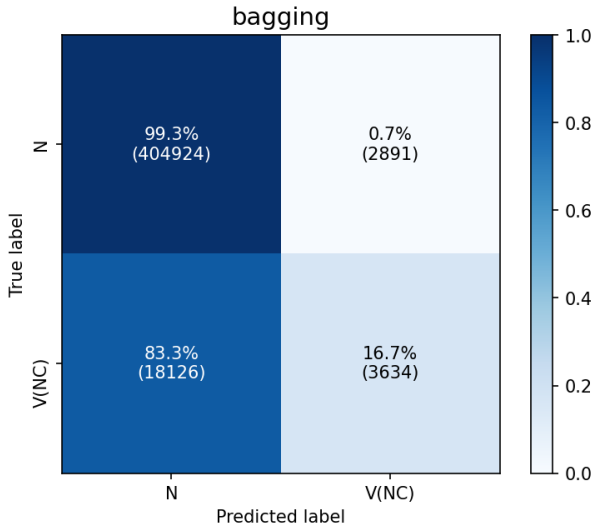


Figure 2: The confusion matrix showing the predictions of the classifier trained with configuration C_2.

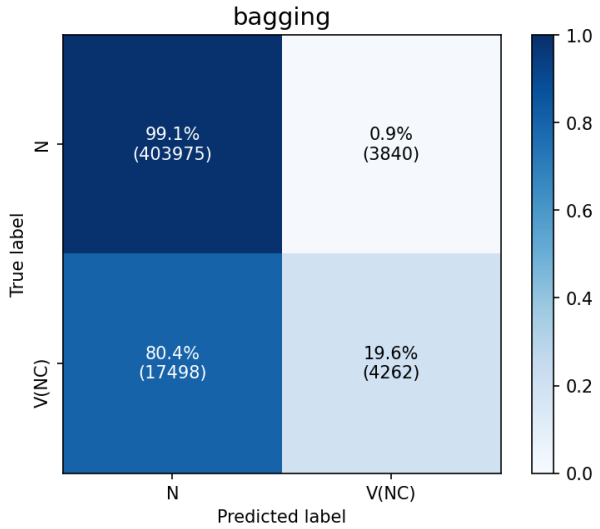


Figure 3: The confusion matrix showing the predictions of the classifier trained with configuration C_3.

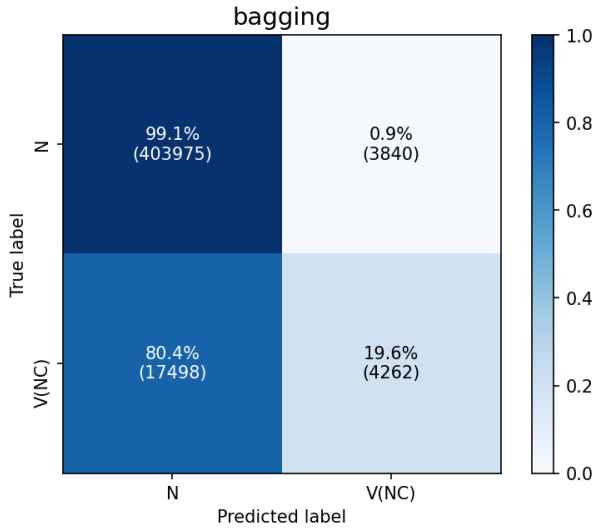


Figure 4: The confusion matrix showing the predictions of the classifier trained with configuration C_4.

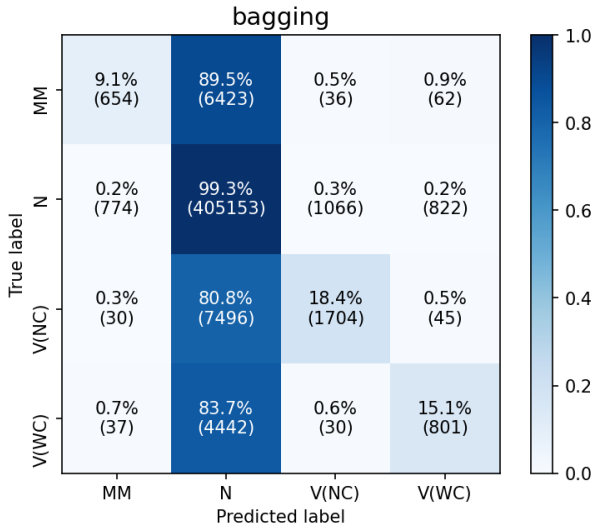


Figure 5: The confusion matrix showing the predictions of the classifier trained with configuration C_6.

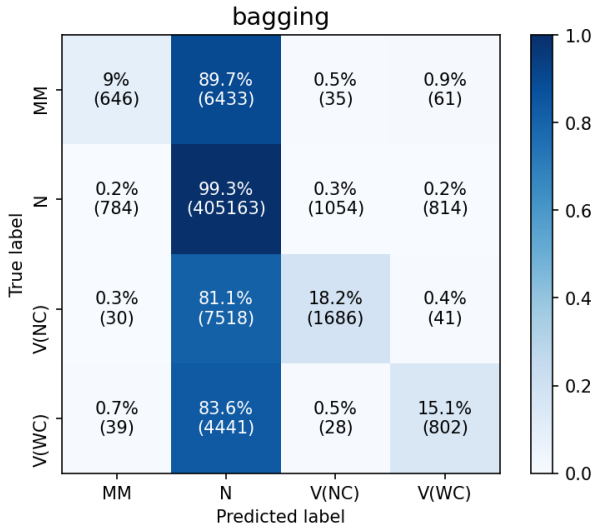


Figure 6: The confusion matrix showing the predictions of the classifier trained with configuration C_7.

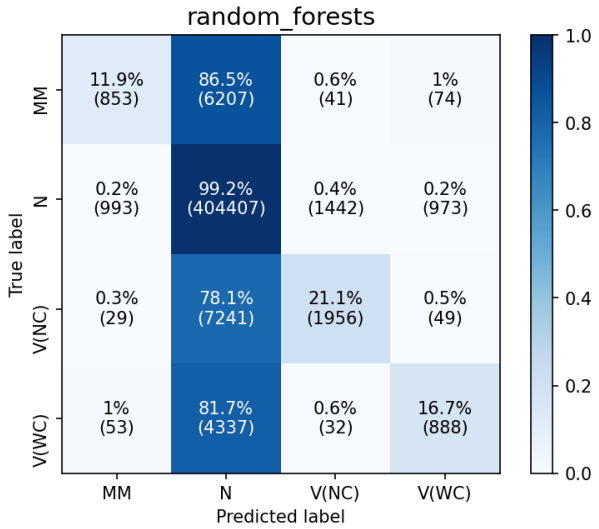


Figure 7: The confusion matrix showing the predictions of the classifier trained with configuration C_8.

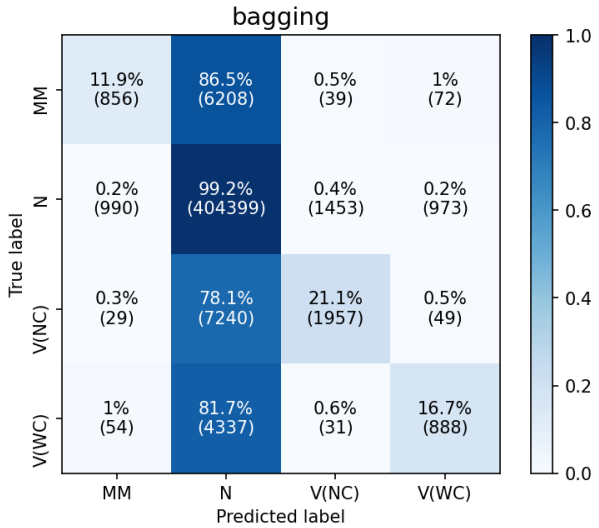


Figure 8: The confusion matrix showing the predictions of the classifier trained with configuration C_9.

This page is intentionally left blank.

Appendix D - Results for the Temporal Window Experiments

This Appendix contains the performance metric values obtained after rerunning the best configurations shown in Section 4.3 with three temporal sliding windows: ten years, five years, and a variable-length window that includes every year before the testing one. An additional figure showing how the performance metrics evolve over time is also presented for each configuration. All configurations in this Appendix are specified in Table 4.5. Note that C_4, C_7, and C_8 are omitted from this temporal analysis as they refer to duplicate pairs of target labels and performance metrics. The results for C_1 are shown in Figure 4.3 and Table 4.7 of this previous section. All figures and tables were generated by the **plot_temporal_windows_results** script documented in Appendix 6.

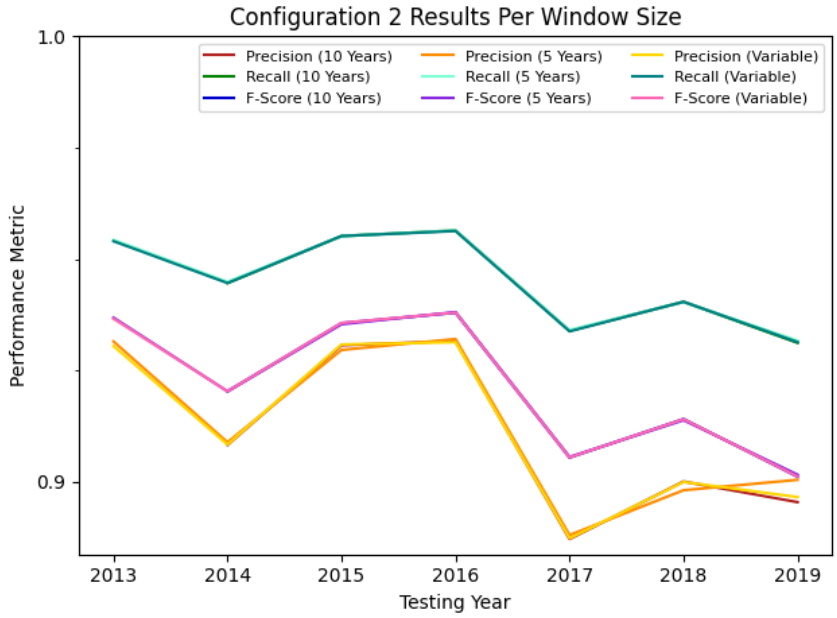


Figure 9: The evolution of the performance metrics for each window size along each testing year for configuration C_2.

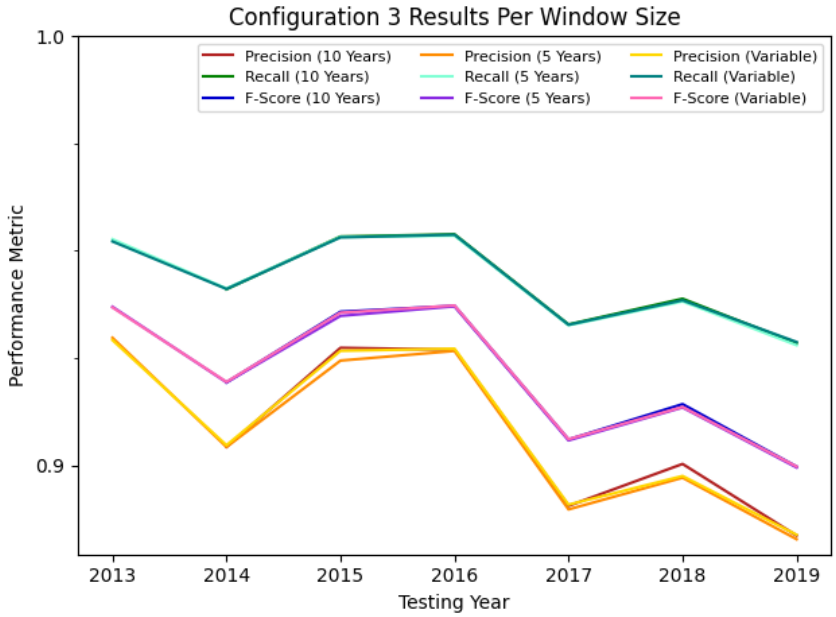


Figure 10: The evolution of the performance metrics for each window size along each testing year for configuration C_3.

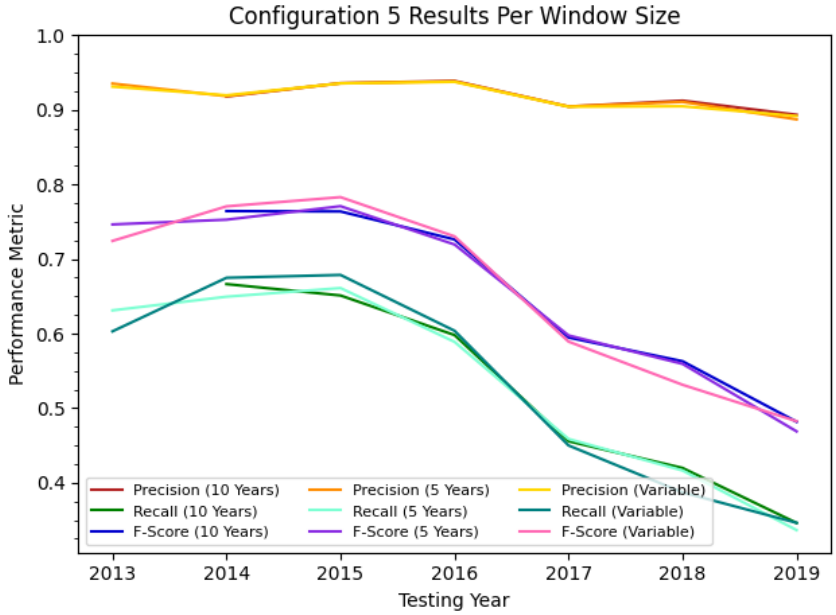


Figure 11: The evolution of the performance metrics for each window size along each testing year for configuration C_5.

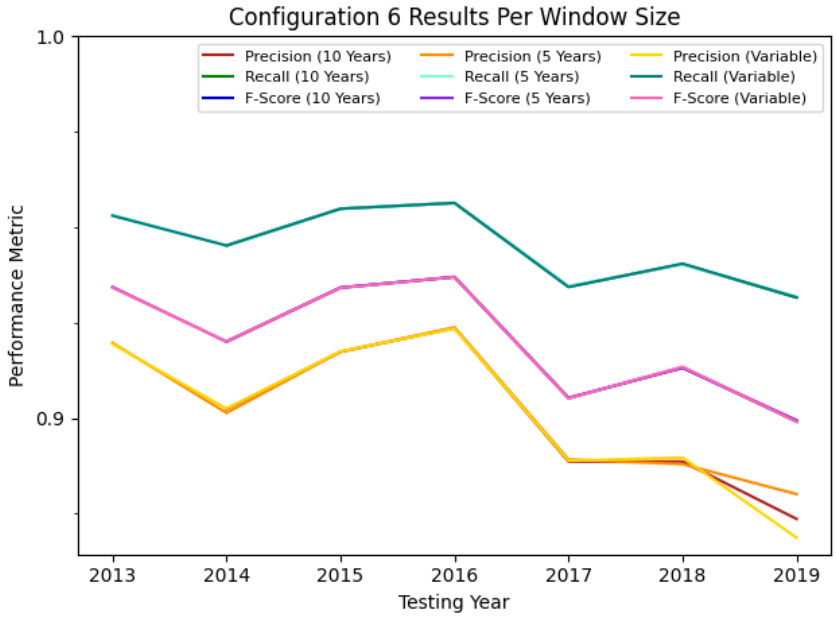


Figure 12: The evolution of the performance metrics for each window size along each testing year for configuration C_6.

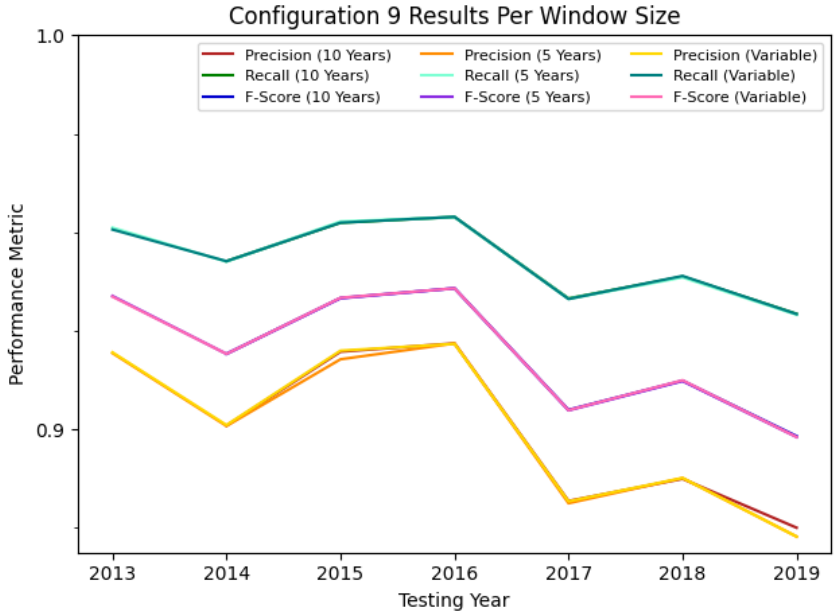


Figure 13: The evolution of the performance metrics for each window size along each testing year for configuration C_9.

Window	Training	Testing	Training %	Precision	Recall	F-score
10	2004-2013	2014	64%	0.9083	0.9446	0.9203
10	2005-2014	2015	78%	0.9306	0.9552	0.9356
10	2006-2015	2016	77%	0.9316	0.9564	0.9380
10	2007-2016	2017	91%	0.8872	0.9339	0.9054
10	2008-2017	2018	96%	0.9001	0.9404	0.9141
10	2009-2018	2019	96%	0.8954	0.9312	0.9012
5	2008-2012	2013	62%	0.9315	0.9543	0.9369
5	2009-2013	2014	63%	0.9089	0.9449	0.9203
5	2010-2014	2015	76%	0.9296	0.9551	0.9354
5	2011-2015	2016	76%	0.9321	0.9564	0.9381
5	2012-2016	2017	90%	0.8881	0.9341	0.9056
5	2013-2017	2018	95%	0.8981	0.9404	0.9138
5	2014-2018	2019	95%	0.9005	0.9317	0.9016
Variable	2002-2012	2013	64%	0.9305	0.9540	0.9366
Variable	2002-2013	2014	64%	0.9084	0.9446	0.9203
Variable	2002-2014	2015	78%	0.9308	0.9552	0.9357
Variable	2002-2015	2016	78%	0.9313	0.9563	0.9380
Variable	2002-2016	2017	91%	0.8873	0.9338	0.9055
Variable	2002-2017	2018	96%	0.9000	0.9404	0.9141
Variable	2002-2018	2019	96%	0.8966	0.9314	0.9012

Table 2: The performance metric values for configuration C_2 using the three temporal sliding windows.

Window	Training	Testing	Training %	Precision	Recall	F-score
10	2004-2013	2014	64%	0.9043	0.9411	0.9193
10	2005-2014	2015	78%	0.9274	0.9533	0.9358
10	2006-2015	2016	77%	0.9270	0.9539	0.9372
10	2007-2016	2017	91%	0.8906	0.9329	0.9060
10	2008-2017	2018	96%	0.9003	0.9388	0.9143
10	2009-2018	2019	96%	0.8836	0.9285	0.8997
5	2008-2012	2013	62%	0.9297	0.9527	0.9370
5	2009-2013	2014	63%	0.9043	0.9412	0.9194
5	2010-2014	2015	76%	0.9244	0.9532	0.9348
5	2011-2015	2016	76%	0.9267	0.9535	0.9371
5	2012-2016	2017	90%	0.8897	0.9327	0.9058
5	2013-2017	2018	95%	0.8971	0.9382	0.9135
5	2014-2018	2019	95%	0.8828	0.9280	0.8995
Variable	2002-2012	2013	64%	0.9292	0.9522	0.9368
Variable	2002-2013	2014	64%	0.9047	0.9411	0.9194
Variable	2002-2014	2015	78%	0.9267	0.9531	0.9356
Variable	2002-2015	2016	78%	0.9272	0.9538	0.9373
Variable	2002-2016	2017	91%	0.8908	0.9328	0.9061
Variable	2002-2017	2018	96%	0.8975	0.9385	0.9136
Variable	2002-2018	2019	96%	0.8838	0.9288	0.8997

Table 3: The performance metric values for configuration C_3 using the three temporal sliding windows.

Window	Training	Testing	Training %	Precision	Recall	F-score
10	2004-2013	2014	64%	0.9179	0.6664	0.7642
10	2005-2014	2015	78%	0.9358	0.6511	0.7637
10	2006-2015	2016	77%	0.9387	0.5980	0.7262
10	2007-2016	2017	91%	0.9043	0.4557	0.5947
10	2008-2017	2018	96%	0.9123	0.4198	0.5629
10	2009-2018	2019	96%	0.8934	0.3461	0.4818
5	2008-2012	2013	62%	0.9352	0.6311	0.7464
5	2009-2013	2014	63%	0.9184	0.6495	0.7527
5	2010-2014	2015	76%	0.9356	0.6611	0.7708
5	2011-2015	2016	76%	0.9381	0.5889	0.7194
5	2012-2016	2017	90%	0.9045	0.4588	0.5975
5	2013-2017	2018	95%	0.9107	0.4164	0.5594
5	2014-2018	2019	95%	0.8872	0.3365	0.4689
Variable	2002-2012	2013	64%	0.9311	0.6030	0.7242
Variable	2002-2013	2014	64%	0.9195	0.6748	0.7706
Variable	2002-2014	2015	78%	0.9355	0.6787	0.7829
Variable	2002-2015	2016	78%	0.9375	0.6038	0.7304
Variable	2002-2016	2017	91%	0.9041	0.4500	0.5892
Variable	2002-2017	2018	96%	0.9047	0.3871	0.5314
Variable	2002-2018	2019	96%	0.8917	0.3463	0.4825

Table 4: The performance metric values for configuration C_5 using the three temporal sliding windows.

Window	Training	Testing	Training %	Precision	Recall	F-score
10	2004-2013	2014	64%	0.9023	0.9451	0.9200
10	2005-2014	2015	78%	0.9173	0.9548	0.9342
10	2006-2015	2016	77%	0.9236	0.9563	0.9369
10	2007-2016	2017	91%	0.8886	0.9344	0.9052
10	2008-2017	2018	96%	0.8886	0.9403	0.9132
10	2009-2018	2019	96%	0.8736	0.9316	0.8991
5	2008-2012	2013	62%	0.9197	0.9532	0.9343
5	2009-2013	2014	63%	0.9013	0.9452	0.9200
5	2010-2014	2015	76%	0.9174	0.9548	0.9341
5	2011-2015	2016	76%	0.9238	0.9564	0.9370
5	2012-2016	2017	90%	0.8891	0.9346	0.9053
5	2013-2017	2018	95%	0.8879	0.9402	0.9131
5	2014-2018	2019	95%	0.8801	0.9318	0.8994
Variable	2002-2012	2013	64%	0.9194	0.9530	0.9342
Variable	2002-2013	2014	64%	0.9024	0.9452	0.9200
Variable	2002-2014	2015	78%	0.9175	0.9548	0.9341
Variable	2002-2015	2016	78%	0.9234	0.9563	0.9369
Variable	2002-2016	2017	91%	0.8888	0.9343	0.9052
Variable	2002-2017	2018	96%	0.8896	0.9404	0.9134
Variable	2002-2018	2019	96%	0.8686	0.9316	0.8990

Table 5: The performance metric values for configuration C_6 using the three temporal sliding windows.

Window	Training	Testing	Training %	Precision	Recall	F-score
10	2004-2013	2014	64%	0.9010	0.9427	0.9192
10	2005-2014	2015	78%	0.9197	0.9524	0.9333
10	2006-2015	2016	77%	0.9218	0.9539	0.9358
10	2007-2016	2017	91%	0.8818	0.9332	0.9050
10	2008-2017	2018	96%	0.8875	0.9388	0.9124
10	2009-2018	2019	96%	0.8751	0.9293	0.8983
5	2008-2012	2013	62%	0.9194	0.9512	0.9339
5	2009-2013	2014	63%	0.9009	0.9426	0.9192
5	2010-2014	2015	76%	0.9178	0.9528	0.9333
5	2011-2015	2016	76%	0.9218	0.9539	0.9358
5	2012-2016	2017	90%	0.8813	0.9331	0.9048
5	2013-2017	2018	95%	0.8877	0.9386	0.9123
5	2014-2018	2019	95%	0.8728	0.9291	0.8981
Variable	2002-2012	2013	64%	0.9195	0.9507	0.9337
Variable	2002-2013	2014	64%	0.9011	0.9427	0.9192
Variable	2002-2014	2015	78%	0.9200	0.9525	0.9334
Variable	2002-2015	2016	78%	0.9217	0.9539	0.9358
Variable	2002-2016	2017	91%	0.8818	0.9331	0.9049
Variable	2002-2017	2018	96%	0.8877	0.9389	0.9125
Variable	2002-2018	2019	96%	0.8728	0.9293	0.8981

Table 6: The performance metric values for configuration C_9 using the three temporal sliding windows.