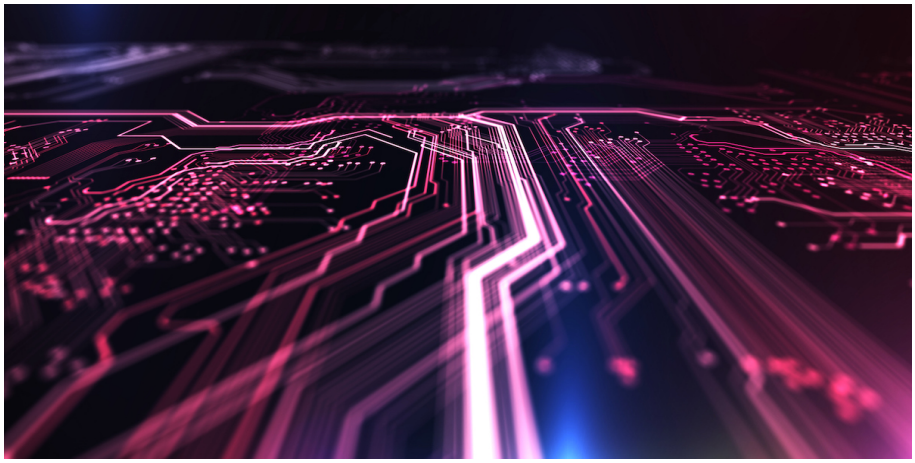
# INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

**Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores**

# Software Weaknesses Detection using Static-code Analysis and Machine Learning Techniques


**Sana Conté**


Licenciatura em Engenharia Informática, Redes e Telecomunicações

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador (es) : Doutor Nuno Leite
Doutor José Simão

Júri:

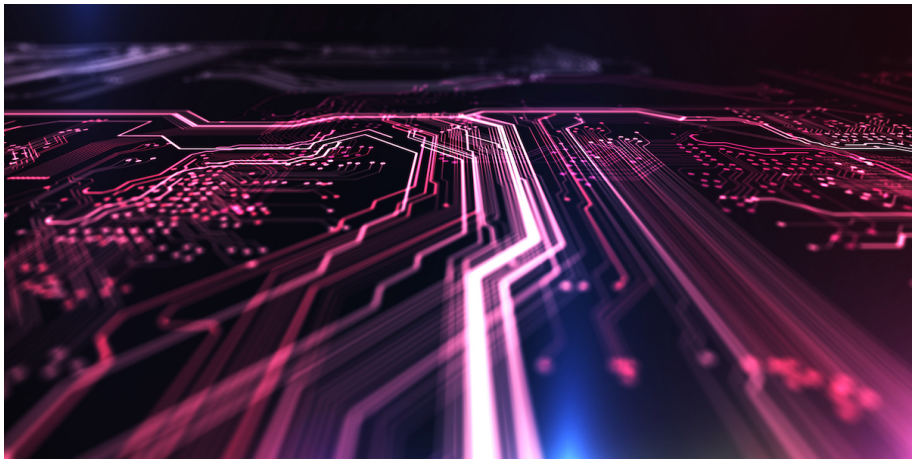
Presidente: Doutor Tiago Miguel Braga da Silva Dias

Vogais: Doutora Cátia Raquel Jesus Vaz
Doutor Nuno Miguel da Costa de Sousa Leite

**Setembro, 2023**

# INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

**Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores**

# Software Weaknesses Detection using Static-code Analysis and Machine Learning Techniques


**Sana Conté**


Licenciatura em Engenharia Informática, Redes e Telecomunicações

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador (es) : Doutor Nuno Leite
Doutor José Simão

Júri:

Presidente: Doutor Tiago Miguel Braga da Silva Dias

Vogais: Doutora Cátia Raquel Jesus Vaz
Doutor Nuno Miguel da Costa de Sousa Leite

**Setembro, 2023**

# Acknowledgments

Words cannot adequately convey my gratitude to my supervisors, Dr. Nuno Leite and Dr. José Simão, who made this work possible. Their invaluable patience and feedback were instrumental in guiding me through every stage of writing this work. Your guidance and advice supported me through all phases of crafting this project. Your availability and encouragement were of immeasurable value in completing this dissertation.

I also couldn't have embarked on this journey without my defense committee, whose knowledge and expertise were generously shared. Furthermore, this undertaking would not have been achievable without the generous support from the Instituto Superior de Engenharia de Lisboa.

I would like to extend special thanks to my entire family for their unwavering support and understanding over the past few years. Your prayers for me were what kept me going this far... Thank you for everything!

I would also like to express my gratitude to my colleagues for the cherished moments we've spent together throughout my academic journey, some of whom have become lifelong friends.

Finally, I want to extend my gratitude to all those individuals who have not been mentioned above but have played a direct or indirect role in bringing this work to completion. To each and every one of them, my heartfelt thanks!

# Abstract

Software industry plays an essential role in modern world in almost all fields. Vulnerabilities are predominant in software systems and can result in a negative impact to the computer security. Although there are tools to detect vulnerable code, their accuracy and efficacy is still a challenging research question. To define features that identify vulnerabilities, many existing solutions require hard work from human experts. The constant increasing number of revealed security vulnerabilities have become an important concern in the software industry and in the field of cybersecurity, implying that the current approaches for vulnerability detection demand further improvement. This has motivated researchers in the software engineering and cybersecurity communities to apply machine learning for patterns recognition and characteristics of vulnerable code. Following this research line, this work presents a machine learning based vulnerability detection system that uses static-code analysis to extract dependencies in the code and build data features from these. The dataset was collected from the National Vulnerability Database (NVD) and test cases NIST SAMATE project and contains Java code as selected target programming language with Null pointer deference and command injections vulnerabilities as selected weaknesses. The data samples were generated from the source code of the vulnerable files by utilizing a control flow graph (CFG) to extract features. Data-flow analysis techniques were also used for feature extraction. Experimental results demonstrate that our tool can achieve significantly fewer false negatives (with a reasonable number of false positives) compared to other approaches. We further applied the tool to real software products and were able to identify vulnerabilities, despite the number of false positives.

**Keywords:** control flow graph, Data-flow analysis, feature extraction, machine learning, reaching definitions, static-code analysis, vulnerability detection.

# Resumo

A indústria de *software* desempenha um papel essencial no mundo moderno em quase todos os domínios. As vulnerabilidades são predominantes nos sistemas de software e podem resultar num impacto negativo na segurança informática. Embora existam ferramentas para detetar códigos vulneráveis, sua precisão e eficácia ainda é uma questão de pesquisa desafiante. Para definir mecanismos que identificam vulnerabilidades, muitas soluções existentes requerem trabalho árduo dos especialistas. O constante aumento do número de vulnerabilidades reveladas tornou-se uma preocupação importante na indústria de software e no campo da cibersegurança, o que implica que as atuais abordagens para a deteção de vulnerabilidades exigem melhorias adicionais. Isso tem motivado investigadores nas comunidades de engenharia de *software* e segurança cibernética a aplicar aprendizagem automática para reconhecimento de padrões e características de códigos vulneráveis. Seguindo esta linha de pesquisa, este trabalho apresenta um sistema de deteção de vulnerabilidades baseado em aprendizagem automática que usa análise estática de código para extrair dependências no código e construir o conjunto de dados a partir destes. A *dataset* foi recolhida a partir da *National Vulnerability Database (NVD)* e o *SAMATE*. A *dataset* contém códigos fonte *Java* com as vulnerabilidades *Null pointer deference* e *command injections* como alvos selecionados para caso de estudo. A *Control Flow Graph (CFG)* foi utilizada em conjunto com as técnicas de análise estática de código para extração de caracteristicas. Os resultados experimentais demonstram que nossa ferramenta pode alcançar significativamente menos falsos negativos (com um número razoável de falsos positivos) em comparação com outras abordagens. Além disso, aplicamos a ferramenta a produtos de software reais e fomos capazes de identificar vulnerabilidades, apesar do número de falsos positivos.

**Palavras-chave:** análise estática de código, aprendizagem automática, Control Flow Graph, deteção de vulnerabilidades, extração de caracteristicas.

# Contents

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**AST** Abstract Syntax Trees. 11, 12, 41, 44, 45

**CFG** Control Flow Graph. 13, 14, 41, 42, 45
**CI** Command Injection. xv, 44, 55, 57, 60

**DT** Decision Tree. 50

**LR** Logistic Regression. 50, 53

**ML** Machine Learning. xii, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
**MLPC** Multi-layer Perceptron classifier. 50, 53

**NB** Naive Bayes. 50, 53
**NC** Neighbor Classifier. 50, 53
**NPD** Null Pointer Deference. xv, 44, 55, 57, 60
**NVD** National Vulnerability Database. 43, 44

**OWASP** Open Worldwide Application Security Project. 37, 38

**SAMATE** Software Assurance Metrics And Tool Evaluation. 43, 44, 52, 57, 58, 60, 61
**SMOTE** Synthetic Minority Oversampling Technique. 47, 52
**SVM** Support Vector Machine. 29, 30, 53

**UD** Use-Definition. 15, 42

# 1

# Introduction

In the present day, the absence of software security presents a significant challenge for companies and organizations. Vulnerabilities identified within software can result not only in added expenses but can also be exploited by malicious actors, leading to varying degrees of harm to individuals and organizations alike. It is imperative to possess the appropriate tools and techniques for predicting and detecting the vulnerabilities present in software components developed by teams.

Organizations such as OWASP (Open Web Application Security Project) strive to address this issue by releasing informative documents that raise awareness among developers and focus on web application security. The most recent standard, the 'Top 10 Web Application Security Risks' published in 2021, documents the most crucial vulnerabilities identified in software. There are several approaches, including both static and dynamic analysis of the source code [18]. However, in recent times, machine learning has been used in numerous ways to develop enhanced models and tools.

Since a lot of tools depend on human experts to identify features that might be susceptible to vulnerabilities, they tend to be subjective and demand extensive manual effort. Consequently, there is a preference for leveraging machine learning algorithms to automatically acquire vulnerability-related features. This enables the algorithms to subsequently recognize prevalent patterns of vulnerable code and alert the user. Using tools that conduct static code analysis assists developers and organizations in identifying vulnerabilities within production code.

More recently, the application of machine learning techniques to software security has been the subject of intense investigation. Despite the existence of numerous static vulnerability detection systems and studies aimed at this objective, they vary from open-source tools to commercial products and to academic research projects.

One of the primary challenges in identifying vulnerabilities lies in the difficulty of extracting features that accurately describe vulnerable code and distinguish it from secure code. There is a wide variety of different approaches of detecting vulnerabilities [80], such as vulnerability prediction models based on software metrics, where more high-level features are used like: code complexity; anomaly detection models that refer to the problem of describing normal and expected behavior and detecting deviations from it; and vulnerable code pattern recognition, in which features of the code itself are created that are typical for vulnerabilities—such as the absence of input validation before execution. This last approach is widely applied in practice and and will also constitute a central focus of this research.

## 1.1 Motivation

Without computer software, modern life would not function as it does. Software has become an essential requirement across various sectors of society, encompassing healthcare, energy, transportation, public safety, education, entertainment, and more. Constructing software that is both safe, reliable, and secure is undoubtedly a complex endeavor. Mistakes made by software architects and engineers can readily lead to software vulnerabilities, and the potential consequences of such vulnerabilities can be severe. A vulnerability such as ransomware could potentially disrupt hospitals and transportation systems, leading to hundreds of millions of dollars in damages [60]. Sometimes, even a minor bug in the code can be enough to create a significant vulnerability, rendering the system susceptible to attacks [22]. An illustrative example is the Heartbleed vulnerability [94] which was discovered in the cryptographic library OpenSSL and impacted billions of internet users. Cyberattacks are increasing significantly, posing threats to governments, businesses, and even civilians, resulting in billions of dollars in losses annually. The number of the new entries in the Common Vulnerabilities and Exposures (CVE) database has been rising. The difficulty of finding vulnerabilities causes strong demands for new methods to help analysis and detect vulnerabilities in software.

### 1.1.1 Why the chosen research problem is interesting and relevant

Numerous attacks outlined earlier can be identified through code auditing. While acknowledged as one of the most potent defense strategies [57], these audits are time-consuming, expensive, and consequently conducted infrequently. Code auditing demands security expertise that many developers do not possess. Consequently, security reviews are frequently conducted by external security consultants, which contributes to the overall cost. Double-audits (auditing the code twice) are often highly recommended because new security errors are frequently introduced even as old ones are being corrected.

The advantage of static analysis is that it can identify all potential security violations without the need to execute the application. The use of machine learning obviates the requirements for the features detection from source code to be accessible and reinforces the detection of vulnerabilities based static code analysis and machine learning. Our approach can be viewed as efficient and less time-consuming when compared to code auditing. It can be scalable in order to be applied to other programming languages and other vulnerabilities such as enabling the analysis of object-oriented programming languages codes with common vulnerabilities.

This approach can be regarded as distinctive due to its ability to detect vulnerabilities with high precision and highlight the specific sections of the code that are suspected to be vulnerable. This combination of scalability and precision can enable our analysis to find all vulnerabilities matching a specification within the small part of the code that is analyzed statically and efficiently. In contrast, the previous practical approach (code review) is typically time-consuming. Without a precise analysis, the previous approach could lead to inadvertently introduce more security weaknesses or flaws while attempting to fix or address existing vulnerabilities.

### 1.1.2 Software vulnerability detection – Challenges

Vulnerability detection is a multi-step and time-consuming process. It involves identifying vulnerabilities, comprehending their impact, assessing associated risks, and determining the priority of risks to address.

The existence of vulnerabilities in software is inevitable [58], as writing secure code is a challenging task that demands extensive expertise. Given that humans are prone to errors, even experienced and skilled developers can make programming mistakes that result in serious repercussions for information security. Detecting software vulnerabilities at an earlier stage helps to mitigate these consequences.

Open-source software has gained extensive utilization across various industries due to its accessibility and flexibility. However, it also introduces potential software security concerns. Programmers often make errors during the code development process, which can result in the creation of software with vulnerabilities. A software vulnerability is a defect in the software design that can be exploited by an attacker in order to obtain some privileges in the system. With the widespread use of open-source software and code reuse practices such as GitHub forks, vulnerable code can rapidly spread from one project to another. Even with substantial time dedicated to searching for vulnerabilities, developers can never be completely certain that their system is one hundred percent secure. Meanwhile, an attacker only needs to discover a single exploitable vulnerability to cause damage, such as crashing a program or exposing sensitive information.

The process of identifying vulnerabilities should be automated using an approach that is not only accurate but also considerably faster than manual code detection. The approach should not require subjective features that are manually defined, but rather learn features from real-life code and automatically adjust to new challenges. A system of this nature would assist human experts by diminishing or eliminating the requirement for the most time-consuming and error-prone tasks involved in vulnerability detection. With a low false positive rate in test results, such system would offer significant benefits to developers by identifying potential vulnerabilities directly in their source code.

In fact, machine learning techniques are not yet widely used, compared to static analysis and other traditional approaches. Studies and investigations have been conducted in the field of machine learning to detect features related to vulnerabilities, however, a significant portion of these studies primarily focus on synthetic code examples, use a very small code base as a dataset, or are only relevant to a small set of projects.

For these reasons, this work aims to make a contribution to the field by presenting a system that utilizes static code analysis along with machine learning techniques to identify critical software security vulnerabilities.

## 1.2 Research Statement and Contribution

The process of vulnerability detection should be automated to a considerable extent, offering a significant level of precision and speed compared to manual source code analysis. It should acquire features from actual vulnerable code and autonomously

identify vulnerabilities. Such a tool would assist human experts by reducing or eliminating the need for time-consuming and error-prone tasks involved in vulnerability detection. With a low rate of false positives in test results, the tool could prove highly beneficial for developers by detecting potential vulnerabilities within the source code.

In fact, machine learning techniques are not yet widely adopted, in contrast to static analysis and other traditional approaches. The application of machine learning techniques to software security has been a subject of intensive investigation, particularly for vulnerability feature detection. Nevertheless, numerous studies focus on synthetic examples. There are several programming languages that have not received substantial attention so far. Different techniques have been applied in source code vulnerability detection, yet certain approaches may not address all issues or may not be suitable for all types of vulnerabilities that have arisen.

## Our contributions

In this work, we have explored the use of static-code analysis hybridized with machine learning techniques to detect critical software security vulnerabilities. The primary emphasis of this research is on examining established static code analysis and machine learning methods, along with their potential application scenarios. Subsequently, a prototype tool is implemented, which leverages these techniques to learn vulnerability features from a comprehensive database sourced from SAMATE [76]. The prototype tool specifically targets the Java programming language. This tool serves as a proof of concept, reinforcing the effectiveness of this approach for vulnerabilities detection. Its applicability highlight the possibility of combining static code analysis and machine learning to identify vulnerabilities in source code.

# 2

# Background and Related Work

This section offers essential background information to comprehend the scope of our work. We commence by introducing vulnerability discovery, presenting foundational concepts within the context of machine learning, static analysis, Abstract Syntax Trees (AST), and Control Flow Graphs (CFG), elaborating on these aspects in further detail. Subsequently, we present an overview of the potential vulnerabilities that will be addressed in our study.

## 2.1 Discovering vulnerabilities

European Union Agency for Cybersecurity (ENISA) [92] defines vulnerability in [21] as: *"The existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event compromising the security of the computer system, network, application, or protocol involved"*. Likewise, they are described as *"A weakness of an asset or group of assets that can be exploited by one or more threats, where an asset is anything that has value to the organization, its business operations, and their continuity, including information resources that support the organization's mission"* in the norm ISO/IEC 27005:2008 [43].

Three key factors can essentially lead to a software vulnerability:

- **Design or specification phase**: During the design phase, important architectural decisions are made, including the choice of programming languages, frameworks, and libraries. Poor choices or inadequate consideration of security requirements at this stage can lead to vulnerabilities. For example, selecting a framework with known security issues or not properly isolating components can introduce vulnerabilities.

- **Implementation phase**: The implementation phase of software development is a critical stage where vulnerabilities can either be introduced or mitigated. Secure coding practices, adherence to secure coding standards, thorough testing, and attention to security details are essential to minimize the risk of vulnerabilities in the final product. Regularly updating and patching software components is also crucial for ongoing security.

- **The deployment phase**: The deployment phase of software development can have a significant impact on software vulnerability. This phase involves the installation, configuration, and setup of the software in its intended environment. Properly configuring security settings, access controls, encryption, and monitoring is essential to ensure that the software remains secure in its operational environment. Additionally, ongoing maintenance and patch management are crucial for long-term security.

There are multiple techniques employed for vulnerability discovery, which can be categorized into static analysis, involving the processing of code without execution; dynamic analysis, where code behavior is analyzed during execution; and hybrid techniques that combine elements of both. These three methods are described in the subsequent sections, preceding the presentation of the fundamental concepts of Machine Learning approaches, along with a discussion of certain vulnerabilities in our case study.

## 2.2 Static analysis

In static code analysis, a program is examined structurally without actual execution. In this approach, potential vulnerabilities are identified [2] through techniques like *taint analysis* and *data flow analysis*. Static analysis provide support for code reviews

by highlighting potential vulnerabilities and security weaknesses. Developers can review and address these issues before the code is committed, reducing the likelihood of vulnerabilities making their way into production. Static analysis tools can be used to recognize patterns and coding practices that are indicative of common vulnerabilities. For example, they can identify instances of SQL injection, cross-site scripting (XSS), buffer overflows, and other security issues based on known coding patterns. Static code analysis tools automatically scan the source code of an application, examining each line and component for potential vulnerabilities. This automation makes it possible to analyze large and complex codebases efficiently. This method holds certain advantages, including enabling faster inspection cycles for fixes, and it is relatively efficient. However, it also has limitations. For instance, it cannot uncover vulnerabilities introduced in the runtime environment [2]. Additionally, due to imperfect models, static analysis often generates false positives, requiring human intervention.

Within static analysis, various approaches can be used, including rule matching, pointer analysis, taint analysis, lexical analysis, model checking, data-flow analysis, and dependency analysis [8, 59]. The concept behind taint analysis is that any variable that can be altered by the user, either directly or indirectly, should be regarded as tainted, as it holds the potential to develop into a security vulnerability. Variables that stem from tainted variables also inherit the tainted status.

Lexical analysis involves scanning the source code to identify patterns or abstract syntax. Overall, lexical analysis is a critical step in the compilation process, as it transforms human-readable source code into a format that can be processed by the subsequent stages of a compiler or interpreter. This process simplifies the task of understanding and manipulating the code's structure and semantics. Data flow analysis involves examining how data values are manipulated, transformed, and transferred between variables and functions throughout the code.

The first static analysis tools that were introduced were basic program checkers [49]. These tools were not highly advanced and often consisted of simple commands like grep or find, relying on a lexical analysis approach. There exists an extensive list of vulnerability scanners employed in practical settings, including tools like PScan [70], Microsoft PREfast [53], and JSLint [48], which enforce sound programming practices. These tools are invaluable, particularly in development environments. However, they fall short when it comes to identifying more intricate and nuanced vulnerabilities. These tools were swiftly succeeded by numerous commercial counterparts designed

for various programming languages, all aiming to accomplish automated source code reviews.

The precision of static analysis tools is evaluated through the calculation of the false positive rate, and it is constrained by computational limitations as well as the rate of false positives. The significant occurrence of false positives in numerous tools poses a considerable challenge [52], as these misclassifications require developers to invest substantial time in verifying each one, thereby complicating the identification of actual issues.

## 2.3 Dynamic and hybrid analysis

Another technique that can be employed to discover vulnerabilities is dynamic analysis, wherein computer software is analyzed on a real or virtual processor [2] through the execution and monitoring of programs during runtime. Dynamic analysis tools try to execute the program using various inputs and monitor its behavior, while static analysis tools aim to construct a model of the program's state to identify potential behaviors through a logical process.

Dynamic analysis approaches can be divided into two main categories: concrete and symbolic execution. Noteworthy applications of dynamic concrete execution include: fuzzing [33, 87], wherein malformed input is provided to provoke crashes; and taint-based fuzzing [83], which assesses how the program handles input to determine which portions of the input require modification in subsequent runs.

Dynamic analysis offers certain advantages, including the ability to identify vulnerabilities during runtime and the capability to analyze applications that lack access to the actual code. However, it also presents limitations, such as the absence of adequately trained professionals to conduct dynamic code analysis. Additionally, tracing the vulnerability back to the precise location in the code is more challenging [2].

The major issue with Dynamic analysis is that it demands significantly more computational time and resources, and is often infeasible in many scenarios, particularly when evaluating larger software or collections of software.

Hybrid analysis combines both static and dynamic analysis techniques to enhance the effectiveness of vulnerability detection in code. This approach leverages the strengths of both methods to overcome their individual limitations. During hybrid analysis, the code is subjected to both static analysis, which involves examining the code's structure

without execution, and dynamic analysis, which involves executing and monitoring the code in a runtime environment.

The goal of hybrid analysis is to achieve a more comprehensive and accurate assessment of the code's security posture. Static analysis helps identify potential vulnerabilities by analyzing the code's structure, logic, and potential patterns of misuse. Dynamic analysis, on the other hand, simulates real-world runtime scenarios, which can help uncover vulnerabilities that may only manifest during execution.

By combining these approaches, hybrid analysis aims to achieve a higher level of accuracy in identifying vulnerabilities, reduce false positives and negatives, and provide a more complete understanding of the code's security vulnerabilities. This approach is especially valuable when dealing with complex and large-scale software applications where relying solely on static or dynamic analysis may not provide sufficient coverage.

## 2.4 Representing source code

There are different approaches available for preprocessing code and representing it in a suitable format when applying machine learning to source code. The upcoming sections will describe several commonly employed techniques that are frequently utilized in research studies, and as a result, will be referenced throughout this work.

### 2.4.1 Abstract syntax trees (AST)

An Abstract Syntax Trees (AST) is a tree-based data structure that excellently captures the syntactic arrangement of the source code. Each node within this tree corresponds to a specific construct found within the source code. This structure offers a hierarchical depiction of the source code's composition, as highlighted by Liu [52].

An AST commonly is produced as the result of the syntax analysis phase within in a compiler. Frequently, it operates as an intermediary representation of the program across various stages essential for the compiler's functionality. Notably, in the compilation process, the AST has a considerable impact on how the compiled code is structured, optimized, and executed, making it an important factor in determining the behavior and performance of the compiled program, as noted by Rahul Khinchi [71].

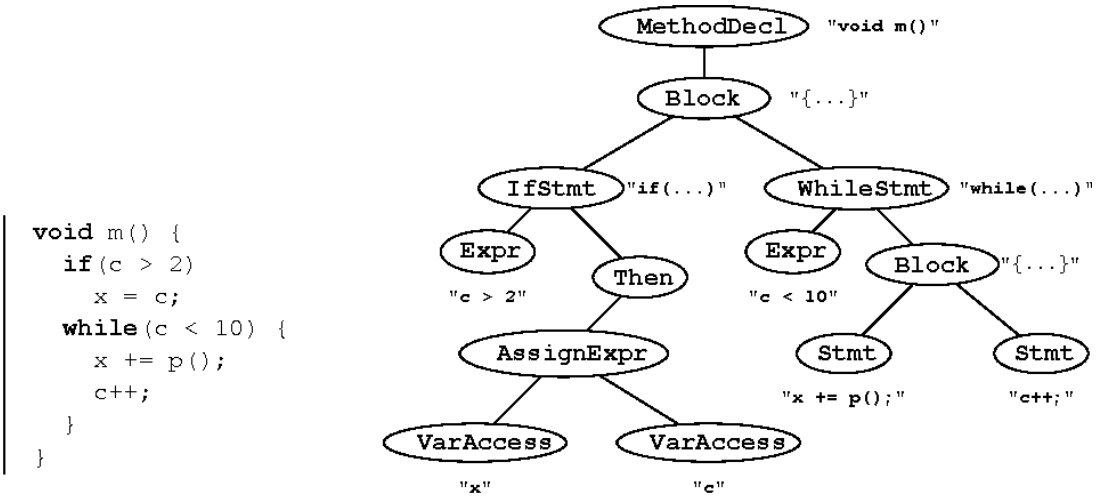



Figure 2.1: Sample Java method and its abstract syntax tree

Figure 2.1 illustrates an example of an abstract syntax tree for a Java method. Each node is depicted as a construct that occurs within the source code.

To build an AST, a specific program analysis involves two fundamental operations: lexical analysis and syntax analysis.

1. **Lexical Analysis (Lexing)** involves transforming the source code of a program into a sequence of tokens (lexemes). Each token is a data structure that signifies a specific type, such as identifiers, keywords, and operators. A program dedicated to performing lexical analysis is typically referred to as a lexer, tokenizer, or scanner. Essentially, a token can be understood as a string endowed with a recognized significance. It is structured as a pair comprising a token name and an optional token value, as outlined by Alfred Aho [3];

2. **Syntax Analysis (Parsing)** entails examining a string of symbols in accordance with the regulations of a formal grammar known as Context-Free Grammar. A program designed for conducting syntax analysis is typically termed a parser, as noted by Douglas Thain [19].

Yamaguchi et al. [22] translated AST graphs into vectors and subsequently applied a natural language processing technique known as latent semantic analysis to identify correlations between established vulnerabilities and possible vulnerabilities present in the code. Through this approach, they successfully identified novel vulnerabilities within multiple open-source projects [22].

The applications of AST are diverse, often finding utility in the realm of static code analysis.

### 2.4.2 Control Flow Graph (CFG)

A Control Flow Graph (CFG) corresponds to a graphical depiction of control flow or computational sequences during program execution. It adopts the structure of a directed graph, where nodes embody basic blocks and edges symbolize potential transitions of control flow from one basic block to another. Within a CFG, in addition to the basic blocks, two specifically designated blocks are present: the *entry* and *exit* blocks. The entry block allows the control flow to enter into the graph, likewise the control flow leaves through the exit block. This is how a control flow graph can depict how different program units or applications process information between different ends in the context of the system [51].

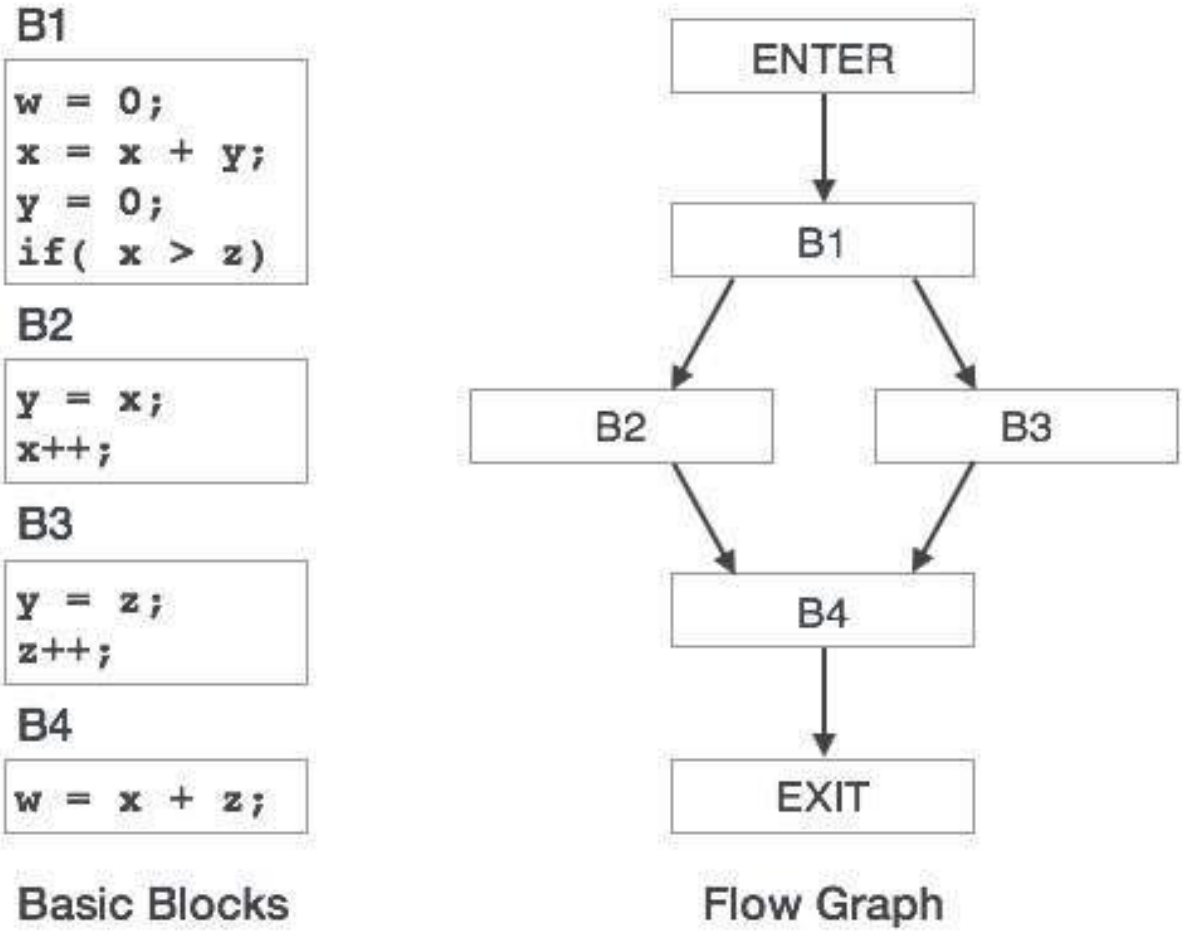



Figure 2.2: Control Flow Graph example

Figure 2.2 illustrates an example of a CFG. Block B1 represents an if statement and branches into two distinct blocks based on the condition (x > z): B2 if the condition is true, or B3 otherwise.

A CFG facilitates the identification of unreachable sections of code within a program,

and it simplifies the detection of syntactic structures such as loops within the graph's structure.

Control flow graph techniques find significant application in data-flow analysis and compiler-related tasks. The forthcoming sections will delve into several techniques employed within data-flow analysis, namely Reaching Definitions, Use-Definition chains, and Taint Analysis. These techniques will also be employed in our own work.

## 2.5 Data Flow Analysis

The subsequent sections discuss techniques used in software analysis and the acquisition of information regarding potential values computed at different junctures within the program. These techniques will consequently find application in the research.

### 2.5.1 Reaching definitions analysis

Reaching definitions analysis [51] constitutes a component of the data-flow analysis approach. Within this method, the definitions that could potentially reach (or be assigned to) a specific point within the code are ascertained. A definition **d** of a variable **v** denotes a statement that allocates a value to the variable **v**. A definition **d** reaches point **p** if a pathway exists from the location immediately following **d** to **p**, wherein **d** remains unaltered (not *killed*) along that pathway.

For example, within the following code blocks:

$$B1 : a = 3$$

$$B3 : a = 4$$

$$B4 : x = a$$

There is no direct pathway between B1 and B4 through B3, meaning that the definition of variable "a" in B1 does not extend to B4 due to the intervening influence of B3, which effectively terminates its reach. A definition of any variable is considered "killed" when a re-assignment occurs between two points along the path.

To compute reaching definitions, IN-OUT Analysis is conducted as described in [3]. For each basic block **B** in a CFG, the following sets are established: $GEN[B]$ represents all definitions generated in block B; $KILL[B]$ represents other definitions that are overlaped (killed) by the definitions generated in block B; $IN[B]$ are all definitions of

the previous blocks, i.e. it turns out to be the union of all $OUT[p]$ for each predecessor **p** of block B $IN[B] = \cup OUT[p]$; $OUT[B]$ are all definitions generated in block B and all definition that have not been "Crushed" in block B. That is, OUT[B] is the union of all definitions in $GEN[B]$ with those that are in $IN[B]$ and not in $KILL[B]$, i.e, $OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$.

```
1
2  for each Block B do {
3      IN[B] = [];
4      OUT[B] = GEN[B];
5  }
6
7  change = true;
8  while change do {
9      change = false;
10     for each Block B do {
11         IN[B] = Up is a predecessor of B OUT[p];
12         oldout = OUT[B];
13         OUT[B] = GEN[B] U (IN[B] - KILL[B]);
14         if(OUT[B] != oldout)
15             change = true;
16     }
17 }
```

Listing 2.1: Algorithm pseudo-code for computing reaching definitions (adapted from [3])

Listing 2.1 presents the reaching definitions algorithm, adapted from [3], tailored to our specific objectives.

The objective is to ascertain, for every point in the program, the assignments that have been executed and persist without being overwritten when execution reaches that specific point via a given pathway. Reaching definitions is used to determine *use-def chains*, which in turn help identify instances of uninitialized or null variables being utilized.

### 2.5.2 Use-definition (UD) Analysis

A Use-Definition (UD) chain consists of a use of a variable, and all the definitions of that variable that can reach that use without any other intervening definitions. They are constructed based on the IN set for a particular definition [51]. Null pointer deference and Command injection vulnerabilities are typically due to the use of potentially vulnerable functions and a lack of sanitization. This is the reason why we will utilize

use-definition chain as well.

## Computing ud-chains

As previously mentioned, we can compute ud-chains from the reaching definitions information. If a use of a variable **A** is preceded in its block by a definition of **A**, then only the last definition of **A** in the block prior to this use reaches the use. Thus the list or ud-chain for this use consists of only this one definition. On the other side, if a use of **A** is preceded in its block **B** by no definition of **A**, then the ud-chain for this use consists of all definitions of **A** in $IN[B]$.

Let U be the statement in question using A:

1. If there are definitions of A within block B prior to statement U, then the last such definition is the only definition of A reaching U. Example:

   Block B:
   $d1 : A = 2$
   $d2 : B = A + 1$
   $d3 : A = 8$
   $d4 : C = A + 5$

   Then:
   <d2, A> = {d1}, The use of A in d2 only consists of d1 exclusively.
   <d4, A> = {d3}, The use of A in d4 only consists of d3 exclusively.

2. If there are no definition of A within block B prior to U, then the definitions of A reaching U are these definitions of A that are in IN[B]. Example:

   Block B:
   $d1 : B = A + 5$

   Lets suppose that IN[B] = {d1, d2, d3, d4, d5} and the definitions of A in set IN[B] are {d2, d5}, then:
   <d1, A> = {d2, d5}, i.e., the use of A in d1 consists of d2 and d5.

### 2.5.3 Taint analysis

Taint analysis is a methodology employed to inject potentially harmful data into a target program, subsequently scrutinizing the trajectory of data propagation to assess program security. This technique was initially introduced in 1998 [73], garnering continuous interest from researchers since its inception. Taint analysis has found extensive application in the detection of information leakage, identification of vulnerabilities, reverse engineering, and other related fields.

The taint analysis process is generally structured into three sequential stages [74]: taint source identification, taint propagation analysis, and taint convergence point detection. Taint sources denote the locations within an application where access to potentially contaminated data occurs. Among these stages, taint propagation analysis stands as the pivotal component of taint analysis methodology. The accuracy of taint propagation analysis is significantly influenced by the taint propagation strategy model.

There are two problems with taint analysis techniques, namely over-taining and under-taining as highlighted by [74]. Over-taining means that data variables with no dependence on the taint source are marked as taint in the process of taint propagation, that is, false positives are generated. Under-taining is that data variables with dependence on the taint source are not marked as taint in the process of taint propagation, that is, false negatives are generated.

Taint analysis identifies every source of user data, encompassing inputs, headers, and subsequent data flows. It traces each fragment of data throughout the system to ensure its proper sanitization prior to utilization. For instance, an application could be susceptible to command injections if it employs unverified external data to execute a command line on the system.

## 2.6 Machine learning (ML)

This section focuses essentially on the techniques used in the development of our tool.

The pioneer of ML, Arthur Samuel, defined ML as a "field of study that gives computers the ability to learn without being explicitly programmed." ML focuses on classification and prediction, based on known properties previously learned from the training data. ML algorithms need a goal (problem formulation) from the domain (e.g., dependent variable to predict). the system to learn from experience and form concepts from examples by extracting patterns from the raw data [34].

A ML approach usually consists of two phases: training and testing. But sometimes, the following steps are performed:

- Identifying class attributes (features) and classes from training data;
- Identifying a subset of the attributes necessary for classification;
- Learning the model using training data;
- Using the trained model to classify the unknown data.

Any Machine Learning method is likely subjective to overfitting, i.e. certain particular features present in a training set which damage the performance for new and not observed examples. This behavior has consequence on error estimation over the training set (the result is overly optimistic). Therefore a separated set of not observed samples is required for an unbiased error estimation [13].

If we want to use error estimation for choosing the best set of hyperparameters (parameters of a model training algorithm defined prior to the learning process) we must use a validation set for this purpose and leave an unseen test set for the final unbiased error estimate. This means that we must split the available data in three parts: the training, validation and test sets.

In fact, for most ML methods, there should be three phases, not two: training, validation, and testing. In the training phase, a machine learning model is built by exposing it to a labeled dataset. The model learns from this data and tries to understand patterns, relationships, and features that will allow it to make predictions or classifications on new, unseen data. After training, it's essential to validate the model's performance. This is typically done using a separate dataset that the model has never seen before. The validation dataset helps assess how well the model generalizes to new data and whether it is overfitting (performing well on the training data but poorly on new data) or underfitting (performing poorly on both training and new data). The testing phase is a crucial step to evaluate the model's performance objectively. A separate dataset, distinct from both the training and validation sets, is used to assess how well the model performs in real-world scenarios. This phase helps estimate the model's accuracy, reliability, and its ability to make accurate predictions on unseen data.[5].

There are three main types of ML approaches: **unsupervised**, **semi-supervised**, and **supervised**. In unsupervised learning problems, the main task is to find patterns, structures, or knowledge in unlabeled data. When a portion of the data is labeled during acquisition of the data or by human experts, the problem is called semi-supervised

learning. The addition of the labeled data greatly helps to solve the problem. If the data are completely labeled, the problem is called supervised learning and generally the task is to find a function or model that explains the data [86].

Once a classification model is developed by using training and validation data, the model can be stored so that it can be used later or on a different data.

The model building is constituted by the following phases:

- Understanding the business: Defining the Data mining problem performed by the project requirements;
- Understanding the data: Data collection and examination;
- Preparing the data: All aspects of data preparation to reach the final dataset;
- Modeling: Applying ML methods and optimizing parameters to fit the best model;
- Evaluation: Evaluating the method with appropriate metrics to verify if business goals are reached;
- Implementation: Varies from submitting a report to a full implementation of the data collection and modeling framework.

Before delving into the algorithms used to create classification models, it's important to grasp the metrics used to evaluate the performance of such models.

According to Somogyi Zolt et al., in their 2021 book titled "Performance Evaluation of Machine Learning Models," the typical metrics employed in model evaluation are as follows [84]:

- **Confusion Matrix**

  In the field of machine learning, a confusion matrix is a table used to describe the performance of a classification model on a set of data for which the true values are known [84]. It's a fundamental tool for understanding the accuracy and behavior of a classification algorithm. The confusion matrix presents the actual and predicted classifications in a tabular format, allowing you to analyze the true positive, true negative, false positive, and false negative predictions made by the model.

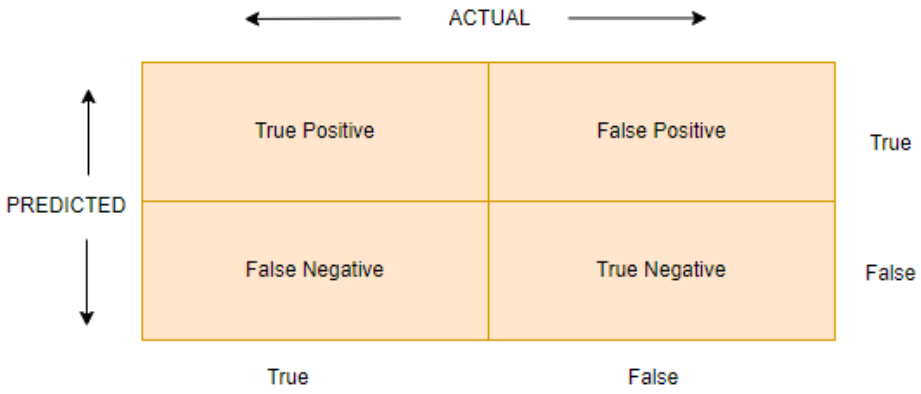



Figure 2.3: Confusion Matrix Example

Figure 2.3 illustrates an example of confusion matrix.

**True Positives (TP)** are positive outcomes in which the model correctly predicts the positive class. In our case, it means: Predicting that the code is vulnerable, and it actually is.

**True Negatives (TN)** are negative outcomes in which the model correctly predicts the negative class. In our case, it means: Predicting that the code is not vulnerable and it actually is not.

**False Positives (FP)** are negative outcomes in which the model incorrectly predicts the positive class. In our case, it means: Predicting that the code is vulnerable, but it is not.

**False Negatives (FN)** are positive outcomes in which the model incorrectly predicts the negative class. For our case, it means: Predicting that the code is not vulnerable, but it is.

From the confusion matrix, various performance metrics can be derived, such as accuracy, precision, recall (sensitivity), specificity, F1-score, etc. These metrics provide a comprehensive understanding of how well the model is performing and can guide further optimization.

- **Precision**

  Is a performance metric that measures the accuracy of positive predictions made by a classification model. It focuses on the ratio of true positive predictions to the total number of instances predicted as positive (both true positives and false positives) [84].

  Mathematically, precision is calculated as:

$$Precision = \frac{TP}{TP + FP} \tag{2.1}$$

  High precision indicates that the model makes fewer false positive errors, meaning that when it predicts a positive instance, it's more likely to be correct. However, optimizing for high precision might result in more false negatives (Type II errors), where positive instances are missed.

- **Recall**

  Also known as sensitivity or true positive rate, is a performance metric that measures the ability of a classification model to correctly identify all relevant instances of a particular class within a dataset. It is particularly important in cases where the consequences of missing positive instances (false negatives) are significant [84].

  Mathematically, recall is calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{2.2}$$

  Recall focuses on the ratio of true positive predictions to the total number of actual positive instances. A high recall indicates that the model is effectively identifying a large portion of the positive instances, which is important in scenarios where false negatives should be minimized, such as in our case, vulnerability detection.

- **f1-score**

  Is a performance metric that combines both precision and recall into a single value. It provides a balanced measure of a model's accuracy, considering both false positives (Type I errors) and false negatives (Type II errors) [84].

  The F1-score is calculated as the harmonic mean of precision and recall:

$$F1Score = \frac{2 * (Recall * Precision)}{(Recall + Precision)} \tag{2.3}$$

The F1-score ranges between 0 and 1, with a higher value indicating better model performance. It is particularly useful when dealing with imbalanced datasets or when the consequences of both false positives and false negatives are important.

- **Area Under the ROC Curve (AUC-ROC)**

  AUC-ROC measures the model's ability to discriminate between positive and negative classes across different threshold values. It ranges from 0 to 1.

  **ROC Curve**: The Receiver Operating Characteristic (ROC) curve is a graphical representation of a model's performance at different classification thresholds. It plots the True Positive Rate (Sensitivity) against the False Positive Rate (1 - Specificity) for different threshold values [84].

  AUC-ROC Interpretation:

  - AUC-ROC values closer to 1 suggest that the model is better at correctly classifying instances and distinguishing between positive and negative classes. AUC-ROC of 1 indicates perfect discrimination between positive and negative classes [84].
  - AUC-ROC values around 0.5 indicate that the model's performance is similar to random guessing.
  - AUC-ROC values below 0.5 indicate that the model's predictions are worse than random guessing.

- **Area Under the Precision-Recall Curve (AUC-PR)** Is a performance metric used in machine learning to evaluate the quality of a binary classification model, especially when dealing with imbalanced datasets [84].

  **Precision-Recall Curve**: The Precision-Recall curve is a graphical representation of a model's performance at different classification thresholds. It plots Precision (the ratio of true positive predictions to all positive predictions) against Recall (the ratio of true positive predictions to all actual positives) for different threshold values.

  **AUC-PR: The Area Under the Precision-Recall Curve (AUC-PR)** is the area under the Precision-Recall curve. Similar to AUC-ROC, it ranges from 0 to 1, where higher values indicate better performance. A higher AUC-PR suggests that the model is better at achieving high precision while maintaining high recall.

  Interpretation of AUC-PR:

  - AUC-PR values closer to 1 suggest that the model has a good balance between precision and recall, effectively classifying positive instances while minimizing false positives.
  - AUC-PR values around 0.5 indicate that the model's performance is similar to random guessing.
  - AUC-PR values below 0.5 indicate that the model's predictions are worse than random guessing.

- **Brier Score** The Brier Score is a performance metric used in machine learning to assess the accuracy of probabilistic predictions made by a model, especially in binary classification tasks. It measures the mean squared difference between predicted probabilities and the actual outcomes [84].

  The Brier Score ranges between 0 and 1, where lower values indicate better model performance. A Brier Score of 0 indicates perfect calibration, meaning that the predicted probabilities match the actual outcomes perfectly.

The following subsections will describe more about sophisticated learning methods that can be applied in vulnerability detection.

### 2.6.1 Neural networks

A neural network consists of neurons, also called units or nodes. They are inspired by the human brain.

The human brain consists of "neurons" working in parallel and exchanging information through their connectors "synapses". These neurons sum up all information coming into them, and if the result is higher than the given potential called action potential, they send a pulse via axon to the next stage [20].

A neural network also consists of simple computing units, called *artificial neurons*, and each unit is connected to the other units via weight connectors. Then, these units calculate the weighted sum of the coming inputs and find out the output using squashing function or activation function [40]. Figure 2.4 shows the block diagram of neuron.

Neurons take input, process it, and pass it on to other neurons present in the multiple hidden layers [40] of the network, till the processed output reaches the Output Layer.

Neural networks are used in many areas. They are predestined for applications in which there is little systematic solution knowledge and a large amount of sometimes

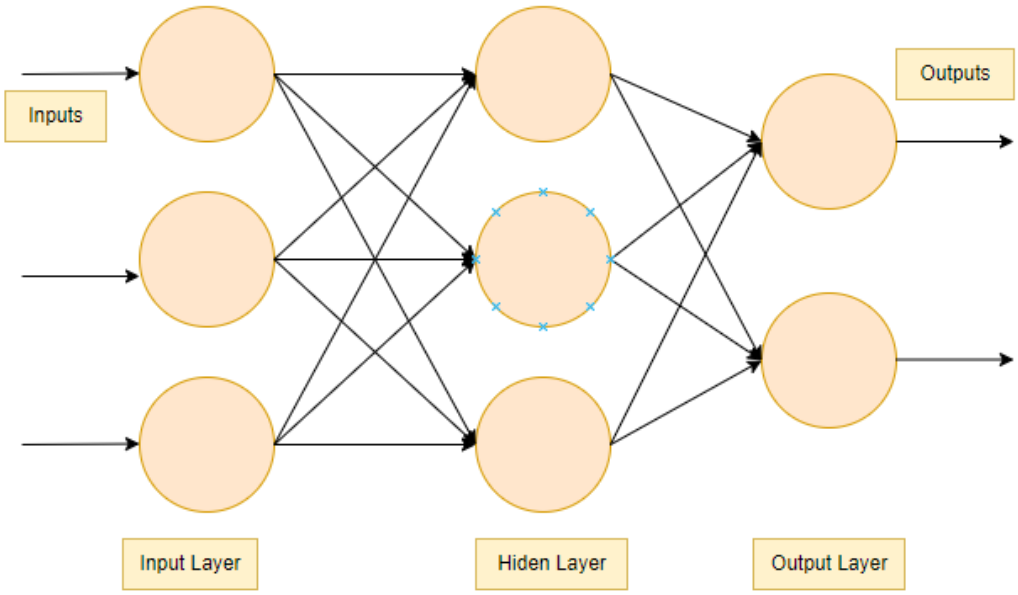



Figure 2.4: Neural network diagram

imprecise input information must be processed to a concrete result [7]. Areas of application can be speech recognition or image recognition. Neural networks can also create simulations and predictions for complex systems and relationships, such as in weather forecasting, medical diagnostics or business processes. Typical applications of neural networks are: Image recognition, Voice recognition, Pattern recognition and more.

Before a neural network can be used for the intended problem or task, it has to be trained. Neural networks learn via supervised learning or unsupervised learning.

Supervised machine learning involves an input variable x and corresponding desired output variable y. The data will be presented in a form of couples (input, desired output), and then the learning algorithm will adapt the weights and biases depending on the error signal between the real output of network and the desired output.

Unsupervised machine learning has input data X and no corresponding output variables. A competitive learning rule is used. A neural network of two layers—an input layer and a competitive layer is used. The input layer receives the available data. The competitive layer consists of neurons that compete with each other (in accordance with a learning rule) for the "opportunity" to respond to features contained in the input data [40].

A neural network is known as **a deep learning or deep neural network** when the newtwork has multiple hidden layers and multiple nodes in each hidden layer [58].

Deep learning is the development of deep learning algorithms that can be used to train and predict output from complex data.

In Deep Learning, the word **"deep"** refers to the number of hidden layers i.e. depth of the neural network. Essentially, every neural network with more than three layers, that is, including the Input Layer and Output Layer can be considered a Deep Learning Model [42].

Generally, it takes more time to train deep learning models. They have higher accuracy than Neural Networks. It gives high performance compared to neural networks.

Deep learning models can be used in a variety of industries, including pattern recognition, speech recognition, natural language processing, and more.

### 2.6.2 Decision Trees

Decision trees [47] are classifiers represented as trees where internal nodes are tests on individual features and leaves are classification decisions. Therefore internal nodes correspond to attributes, which are also known as features or input variables, and leaf nodes correspond to class labels.

Here's how a decision tree is built:

1. **Starting Point**: Begin with all the data at the top node, known as the root node.
2. **Decision Making**: At each step (node), the algorithm decides which feature to use and what value to compare it to. This decision is made by evaluating how well different choices separate the data based on specific criteria:
    - For classification tasks, it might be how "impure" the groups are (measured by Gini impurity or entropy).
    - For regression tasks, it might be how much variance there is (measured by mean squared error).
3. **Data Splitting**: Once the feature and threshold value are chosen, the data is split into two or more subsets based on this decision. Each subset represents a branch of the tree.
4. **Repetition**: Steps 2 and 3 are repeated for each branch, creating new nodes, until a stopping condition is met, like a maximum depth or a minimum number of data points in a leaf.

5. **Leaf Nodes**: The final nodes of the tree, called leaf nodes, contain the predictions or outcomes based on the majority class (in classification) or the average value (in regression) of the data points within that node.

6. **Prediction**: To make predictions for new data, you start at the root node and follow the branches based on the feature values of the new data until you reach a leaf node, which provides the prediction.

In essence, decision trees recursively divide the dataset into subsets using specific criteria at each internal node, creating a tree structure that allows for straightforward decision-making and prediction [50]. The following tree algorithms are worthy of attention: ID3, C4.5, CART, CHAID, and MARS [46, 86].

One advantage of utilizing decision trees over other models is their exceptional interpretability and automatic feature selection. This enables thorough analysis to be conducted on decision trees. However, they can be prone to overfitting if they are not properly pruned or if the tree becomes too deep.

Decision trees are especially well-suited for data characterized by discrete-valued features. Contemporary implementations involve trimming certain branches (those with unexpected information gain), which helps prevent overfitting. This technique is referred to as pruning. Pruning reduces the size of decision trees by eliminating segments of the tree that contribute little to the ability to classify observations.

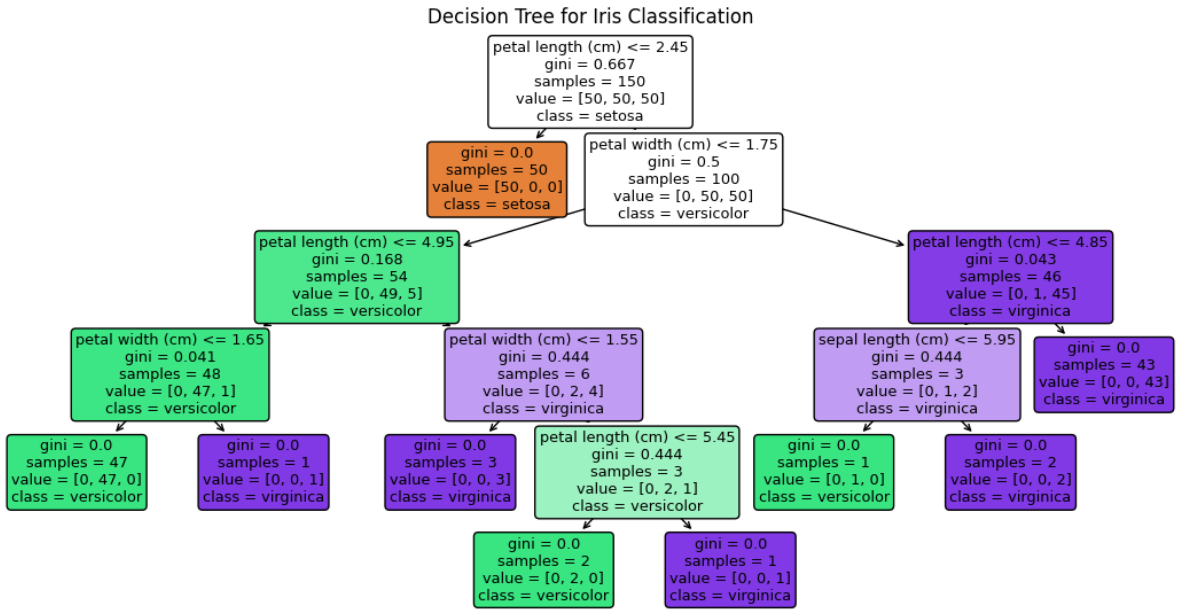



Figure 2.5: Decision tree trained on the iris dataset

The splitting process makes the method highly efficient. When training a decision tree classifier, the algorithm examines the features and determines which splits contain more information.

The decision tree (Figure 2.5) visually illustrates how the dataset is partitioned based on features and feature thresholds, leading to the prediction of iris species at the leaf nodes. Each branch in the tree represents a decision based on a specific feature and threshold value. The tree is built in a way that optimally separates the different iris species based on the available features.

The top node of the tree (called root node) is the starting point for making decisions. In this case, it represents the first feature used for splitting the data. The label on the node indicates which feature is being considered (e.g., petal length (cm) <= 2.45).

Below the root node, you'll see *child nodes*. Each child node represents a possible outcome of the test at the parent node. In figure 2.5 there are usually two child nodes, corresponding to a "True" or "False" outcome of the feature test.

The terminal nodes at the bottom of the tree are called *leaf nodes*. They represent the final predicted class labels. In this example, the leaf nodes have labels like *setosa*, *versicolor*, and *virginica*, indicating the predicted iris species.

The criteria for splitting each node (e.g., Gini impurity or entropy) are not explicitly shown in the figure but are used internally by the decision tree algorithm to determine how to split the data at each node.

The *samples* value in each node represents the number of data points that reach that particular node during the decision-making process. For instance, if the root node has 150 samples, it means the entire dataset was used for that split.

The *value* shows the distribution of target classes (iris species) in that node. For example, if a leaf node has a value of [0, 1, 0], it mean that it predicts one sample as "versicolor" (class 1).

The *gini* index represents the purity of the classification. It ranges between 0 and 1, where 0 and 1 indicate a random distribution of elements among different classes. A Gini index of 0.5 indicates an equal distribution of elements across various classes.

Random forest [54] is a collection of decision trees trained on bootstrapped datasets with a random selection of features. This model is widely adopted for classification due to its resistance to overfitting [65] and the small number of hyperparameters that need to be optimized during the training phase.

### 2.6.3 Naïve Bayes

Naïve Bayes classifiers [37] are straightforward probabilistic classifiers that utilize the Bayes theorem. The name comes from the assumption that input features are independent, though this is rarely true. It stores the prior probability of each class, denoted as $P(C_i)$, and the conditional probability of each attribute value given the class, denoted as $P(V_j \mid C_i)$, in its concept description. It estimates these values by tallying the frequency of class occurrences and attribute values in the training data. Then, assuming that the attributes are conditionally independent, it employs Bayes' rule to calculate the posterior probability of each class for an unknown instance. The classifier predicts the class with the highest posterior probability:

$$P(V_j \mid C_i) = \arg\max P(C_i) \prod P(V_j \mid C_i)$$

Naïve Bayes classifiers can handle any number of independent features, whether they are continuous or categorical. They simplify a high-dimensional density estimation task into a one-dimensional kernel density estimation, based on the assumption of feature independence [5].

Despite their seemingly simple design and oversimplified assumptions, Naïve Bayes classifiers have performed well in many complex real-world scenarios. In 2004, an analysis of the Bayesian classification problem revealed solid theoretical justifications for the seemingly unlikely effectiveness of Naïve Bayes classifiers.[35].

However, Naïve Bayes often struggles to provide accurate estimates for the correct class probabilities, as indicated by [61], which implies that this might not be necessary for many applications. For instance, the Naïve Bayes classifier will produce the right classification result based on the Maximum a Posteriori (MAP) [56] classification rule as long as the predicted class is more likely than any other class, even if the probability estimate for that class is slightly or significantly inaccurate (misclassification of the model, leading to incorrect predictions for specific class). This approach ensures that the overall classifier remains robust enough to overlook significant shortcomings in its underlying naive probability model. While the Naïve Bayes classifier does possess several limitations, it serves as an optimal classifier when the features are conditionally independent given the true class.

### 2.6.4 Logistic regression (LR)

Also referred to as the *logit model*, logistic regression is used for classification and predictive analytics. It estimates the probability of an event occurring, such as whether a person voted or didn't vote, based on a given dataset of independent variables. Because the outcome is a probability, the dependent variable is constrained between 0 and 1. In logistic regression, a logit transformation is applied to the odds, which represents the probability of success divided by the probability of failure [36]. Logistic regression (LR) is a classification algorithm employed to predict a binary outcome based on a set of independent variables.

The Logistic Regression model is a commonly used statistical model primarily employed for classification purposes. This means that when provided with a set of observations, the Logistic Regression algorithm assists in categorizing these observations into two or more distinct classes. Therefore, the target variable is of a discrete nature. This model finds applications in various fields, including machine learning, numerous medical domains, and social sciences. For example, the Trauma and Injury Severity Score (TRISS), a widely used tool for predicting mortality in injured patients, was originally developed by Boyd et al. using logistic regression [10]. Numerous other medical scales employed to assess the severity of a patient's condition have also been created utilizing logistic regression [9, 44, 45, 62].

The Logistic Regression model necessitates the dependent variable to be binary, multinomial, or ordinal in nature. It also requires that observations be independent of each other, meaning they should not arise from repeated measurements. The model assumes linearity of independent variables and log odds. Log odds essentially transform the Logistic Regression model from being probability-based to a likelihood-based model. These log odds are employed to circumvent modeling a variable with a restricted range, such as probability [55].

The success of the Logistic Regression model relies on the sample sizes. Generally, it demands a large sample size to attain high accuracy.

### 2.6.5 Support Vector Machines (SVM)

Support Vector Machine (SVM) is a classification algorithm that operates by identifying a separating hyperplane in the feature space between two classes. This is done in a manner that maximizes the distance between the hyperplane and the nearest data

points of each class. The approach focuses on minimizing the classification risk [88], rather than striving for optimal classification. This algorithm is often used for tasks such as image classification, text categorization, and more.

A SVM constructs a hyperplane or a set of hyperplanes in a high-dimensional or even infinite-dimensional space. These hyperplanes are utilized for various tasks, including classification, regression, and even outlier detection. SVM are a versatile machine learning technique that can effectively handle complex data patterns and are widely used in various domains [78].

The method produces a linear classifier, so its concept description is a vector of weights, $\vec{W}$, a vector of inputs ( training samples) $\vec{X}$ and an intercept or a threshold, **b**. Unlike other linear classifiers, such as Fisher's (1936), SVM uses a kernel function [78] to map training data into a higher-dimensioned space so that the problem is linearly separable. It then uses quadratic programming to set $\vec{W}$ and **b** such that the hyperplane's margin is optimal, meaning that the distance is maximal from the hyperplane to the closest examples of the positive and negative classes. During performance, the method predicts the positive class if hyperplane of the form $\langle \vec{W}, \vec{X} \rangle - b > 0$, and predicts the negative class otherwise [50]. The equation represents internal product, where $\vec{W}$ is a weight vector, $\vec{X}$ is input vector and $b$ is the bias.

SVM works effectively in high dimensional spaces, and it is still efficient in cases where the number of dimensions is greater than the number of samples. The method uses a subset of training points in the decision function (called support vectors), therefore, we can say that it is also memory efficient. It can be versatile in order to allow specifying different kernel function for decision function [78].

It is crucial to choose a good kernel function in order to avoid overfitting when the number of features is much greater than the number of samples, being this a disadvantage of this classifier.

Selecting an appropriate kernel function is of paramount importance to mitigate the risk of overfitting, as highlighted by Webb [65]. This becomes particularly crucial when dealing with situations where the number of features significantly exceeds the number of samples, as this presents a notable drawback of this particular classifier.

## 2.7 Systems

Up to this point, we have been describing the basic concepts necessary for this work. The subsequent section will describe previous works related with vulnerabilities detection at software and attempts of using machine learning. Despite of many different

criteria under which approaches can be compared, the advantages and disadvantages of the previous approaches are described, and our approach is compared with them.

### 2.7.1 Discovering vulnerabilities using data-flow analysis and machine learning

Kronjee et al. introduce an innovative approach to static analysis, wherein they combine data-flow analysis with machine learning techniques to identify vulnerabilities within source code [51].

They had investigated whether machine learning techniques in conjunction with static code analysis can be used to identify insecure code within a particular dynamic programming language, specifically PHP. The focus of the research has been on web application vulnerabilities, specifically SQL injection (SQLi) and Cross-Site Scripting (XSS). The key achievement of this study involves the development of a tool capable of detecting vulnerable PHP code through the utilization of a probabilistic classifier and features extracted via data-flow analysis.

They assembled a dataset from sources such as the National Vulnerability Database (NVD) [64] and Software Assurance Metrics And Tool Evaluation Project (SAMATE) [76] project. This dataset comprised samples of vulnerable PHP code as well as their patched versions, where the vulnerabilities were rectified. They employed data-flow analysis techniques, including reaching definitions analysis, taint analysis, and reaching constants analysis, to extract features from the code samples. Subsequently, they used these features within machine learning to train diverse probabilistic classifiers.

With the results obtained, they demonstrated that using machine learning in combination with features extracted from control-flow graphs and abstract syntax trees can be an effective approach for vulnerability detection in dynamic languages, in particular PHP applications. To demonstrate the effectiveness of the approach, they built a tool called `WIRECAML`, and compared the tool to other tools for vulnerability detection in PHP code.

However, the tool does exhibit a number of limitations, both in aspect of inherent nature (essential part of the tool) and in aspect of technical nature. Due to the complexity of processing arrays of data and, due to the fact that the data set consists of multiple vulnerabilities in different versions of the same application and considering that these application versions share a lot of the same code, the data set can contains duplicate samples.

### 2.7.2 Detecting Software Vulnerabilities with Deep Learning

Wartschinski et al. introduce an approach known as "Vulnerability Detection with Deep Learning on a Natural Codebase" (VUDENC) [89]. This is a vulnerability detection system based on deep learning, designed to autonomously learn features from an extensive compilation of real-world code. The primary purpose of VUDENC was to relieve human experts from the labor-intensive and subjective task of manually determining features for vulnerability detection [89].

VUDENC selected Python as the target programming language for its case study. A large dataset of commits was collected and mined from Github and labeled according to the commit context. The data stems from several hundred real-world repositories containing natural source code and covers seven different types of vulnerabilities, including SQL injections, cross-site scripting and command injections. The data samples were created from the source code of the vulnerable files by taking individual code tokens and their context, allowing for a fine-grained analysis. A word2vec model was used to train on a large corpus of Python code to be able to perform embeddings of code tokens (vector representations of source-code) that preserve semantics, and has been made available as well. Word2vec [93] is a technique for natural language processing (NLP) [63]. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text (a large and structured set of texts).

After pre-processing the raw source code, the datasets for each vulnerability were built by taking every single code token with its context (the tokens before and after it) as one sample and embedding it using the word2vec model. A Long Short Term Memory (LSTM) network was trained on each dataset to detect vulnerable code on the level of individual tokens. The author was able to demonstrate that the experiments with VUDENC achieved, on average, an accuracy of 96.8%, a recall of 83%, a precision of 91% and an F1 score of 87%. These results were very promising and encourage further research in this area [89]. VUDENC was able to highlight the specific areas in code that are likely to contain vulnerabilities and provide confidence levels for its predictions.

The research successfully demonstrate the capability of employing machine learning directly on source code to learn vulnerability-related features through the utilization of LSTM models. VUDENC was specifically designed to operate on Python source code and exhibit proficiency in predicting 7 distinct types of vulnerabilities [89].

### 2.7.3 VulDeePecker A Deep Learning Based System For Vulnerability Detection

Zhen Li et al. presented VulDeePecker [95], the first deep learning based vulnerability detection system. This system aims to relieve human experts from the tedious and subjective work of manually defining features and reduce the false negatives that are incurred by other vulnerability detection systems.

In their work, the authors start by presenting some preliminary principles for guiding the practice of applying deep learning to vulnerability detection because deep learning was not invented for this kind of applications, which means that they needed some guiding principles for applying deep learning to vulnerability detection. They have collected, and made publicly available, a useful dataset for evaluating the effectiveness of VulDeePecker and other deep learning-based vulnerability detection systems that could be developed in the future.

Systematic experiments have showed that VulDeePecker could achieve much lower false negative rate than other vulnerability detection systems. For the 3 software products they experimented with (i.e., Xen, Seamonkey, and Libav), VulDeePecker detected 4 vulnerabilities, which were not reported in the NVD and were "silently" patched by the vendors when they released later versions of these products. In contrast, the other detection systems missed almost all of these vulnerabilities, except that one system detected 1 of these vulnerabilities and missed the other three vulnerabilities.

Despite having a sound approach, the design, implementation, and evaluation of the VulDeePecker exhibited several limitations, which suggested open problems for future research. Their tool's design is constrained to dealing vulnerability detection by assuming that the source code of programs is available. Detecting vulnerabilities in executables presented a distinct and more challenging problem. Their tools' design focuses exclusively on C/C++ programs. Future research needs to be conducted to adapt it to deal with other kinds of programming languages.

According to the authors, the tool has two advantages. First, it does not need human experts to define features for distinguishing vulnerable code and non-vulnerable code. Second, it uses a fine-grained granularity to represent programs, and therefore can pin down the precise locations of vulnerabilities.

# 3

# Architecture

In this section, we will discuss the proposed architecture for the approach taken in this work. This includes the scope, data sources, data representation choices, weaknesses, and the chosen target programming language selected for our case study.

## 3.1 Selecting Java as target programming language

Despite Java is considered a relatively safe language, there are still several ways to attack and access private information if we misuse it, leading to vulnerabilities such as Command injection and cross-site scripting attacks. Fred Long [1] asserts that Java is secure if is used properly, but engineers can misuse it or improperly implement it. A simple programming mistake can leave a java application vulnerable to unauthorized data access, unauthorized updates or even loss of data, and application crashes leading to denial-of-service attacks.

Among all the vulnerabilities detected in Java applications, those arising from **unchecked input** are widely acknowledged as the most prevalent [67].

To exploit unchecked input, an attacker needs to achieve two goals:

- Inject malicious data into application;
- Manipulate applications using malicious codes such as Command injection or Cross-site scripting.

All those types of attacks listed above are made possible by user input that has not been (properly) validated.

This work focuses in source code written in Java. The next subsections will delineate some typical vulnerabilities that are considered in our study case. It's important to note that all the provided examples are deliberately kept simple, solely intended to illustrate the overarching concept, and the actual exploitation of these vulnerabilities is considerably more intricate in practical scenarios.

## 3.2 NULL Pointer Deference vulnerability

A pointer deference [16, 39] is a programming language data variable that references a location in memory. After the value of the location is obtained by the pointer, this pointer is considered dereferenced. It is a widespread vulnerability that occurs whenever an executing program attempts to deference a null pointer [91], i.e., a pointer which references a null location.

*Example Language:* **Java**

```
...
IntentFilter filter = new IntentFilter("com.example.URLHandler.openURL");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
...

public class UrlHandlerReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    if("com.example.URLHandler.openURL".equals(intent.getAction())) {
      String URL = intent.getStringExtra("URLToOpen");
      int length = URL.length();

    ...
    }
  }
}
```

Figure 3.1: Null pointer deference java example

Figure 3.1 depicts a Null pointer dereference example. The code snippet is from an Android application that has registered to handle a URL when sent an intent. The application code exemplified assumes the URL (Uniform Resource Locator) will always be included in the intent. When the URL is not present, the call to $getStringExtra()$ will return null, thus causing a null pointer exception when length() is called.

NULL Pointer Dereference vulnerability can be exploited by hackers to maliciously crash a process to cause a denial of service or execute an arbitrary code under specific conditions. This typical taint-style vulnerability requires an accurate data dependency analysis to trace whether a source is propagated to a sensitive sink without proper sanitization [91]. This vulnerability is mostly used to crash a kernel or process to cause a denial-of-service attack.

NULL Pointer Dereference has been included in "CWE Top 25 Most Dangerous Software Weaknesses" published by the CWE website [17] every year. The ranking scores are calculated by considering both the number of reported vulnerabilities and the potential severity of a vulnerability. An attacker can exploit it indirectly to trigger an arbitrary code execution or bypass authentication [69], in specific situations. For example, CVE-2021-42264 [14] reported that *Adobe Premiere Pro 15.4.1* (and earlier) is affected by a Null pointer dereference vulnerability when parsing a specially crafted file. An unauthenticated attacker could leverage this vulnerability to achieve an application denial-of-service in the context of the current user. Exploitation of this issue requires user interaction in that a victim must open a malicious file.

The following points may be considered as potential mitigations:

- By performing sanity checks on all pointers that could have been modified, nearly all NULL pointer references can be prevented;
- Check the results of the returned value of functions to verify that this value is not NULL before using it;
- Perform input validation on variables and data stores that may receive input from an external source and apply input validation to make sure that they are only initialized to expected values;
- Explicitly initialize all variables and other data stores during declaration or before the first usage.

## 3.3 Command Injection vulnerabilities

Command Injection is a general term for attack types which consist of injecting commands that are consequently executed by the vulnerable application. This type of attacks is considered as a major security threat which in fact, is classified as No. 3 on the 2021 Open Worldwide Application Security Project (OWASP) Top Ten web security risks [68].

Command injection vulnerabilities typically occur when data enters the application from an untrusted source. The data is part of a string that is executed as a command by the application. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

According to the OWASP [90], Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers, etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation.

The impact of command injection attacks may vary from loss of data confidentiality and integrity to unauthorized remote access to the system that hosts the vulnerable application. In particular, an attacker can gain access to resources that he/she does not have privileges to directly accessing them, such as system files that include sensitive data (e.g., passwords). An attacker can perform various malicious actions to the vulnerable system, such as delete files or add new system users for remote access and persistence. An example of a real, infamous command injection vulnerability that clearly describe the threats of this type of code injection was the recently discovered (i.e., disclosed in 2014) Shellshock bug [81].

Command injections vulnerabilities may be present in applications which accept and process system commands from the user input. The purpose of a command injection attack is the injection of an operating system command through the data input to the vulnerable application which in turn executes the injected command (see Figure 3.2).

*Example Language:* **Java**

```
...
String btype = request.getParameter("backuptype");
String cmd = new String("cmd.exe /K \"
  c:\\util\\rmanDB.bat "
  +btype+
  "&&c:\\utl\\cleanup.bat\"")

System.Runtime.getRuntime().exec(cmd);
...
```

Figure 3.2: Command injection vulnerability - Java example

Figure 3.2 portrays a Command Injection Vulnerability situation in Java code excerpt. The following code is taken from an administrative web application designed to enable users to initiate a backup of an Oracle database using a batch-file wrapper around the *rman* utility and subsequently execute a *cleanup.bat* script to remove certain temporary files. The script *rmanDB.bat* accepts a single command-line parameter that specifies the type of backup to be performed. Since access to the database is restricted, the application executes the backup as a privileged user.

The issue here is that the program lacks validation for the `backuptype` parameter provided by the user. Typically, the `Runtime.exec()` function does not execute multiple commands. However, in this case, the program first invokes the `cmd.exe` shell to run multiple commands with a single call to `Runtime.exec()`. Once the shell is invoked, it will execute multiple commands separated by double ampersands `&&`. If an attacker submits a string in the form of `& del c:\dbms\.`, the application will execute this injected command in addition to the commands specified by the program. Due to the application's nature, it runs with the privileges required for interacting with the database. As a result, any injected command by the attacker will also run with these elevated privileges.

There are many types of code injections attacks including Command injections, SQL Injections, Cross Site Scripting, XPath Injections and LDAP (Lightweight Directory Access Protocol) Injection [4]. In this work, we will exclusively deal with command injection attacks. Command injections [4] are prevalent in any application, regardless of the operating system hosting the application or the programming language in which the application is developed. Consequently, they have also been discovered in web applications hosted on web servers, whether they are Windows-based or Unix-based.

The following points may be considered as potential mitigation:

- Use library calls, if possible, rather than external processes to recreate the desired functionality. Ensure that all external commands called from the program are statically created;

- Assign permissions to the software system that prevents the user from accessing/opening privileged files;

- Consider all potentially relevant properties, including length, type of input, when performing input validation.

## 3.4 Proposed Architecture

Up to this point, we have been describing the chosen target programming language selected for our case study. The next section will outline the proposed architecture for the approach we have taken.

We proposed an approach inspired by a research that used data-flow analysis and machine learning to detect SQLi and XSS vulnerabilities in PHP software code [51]. In the proposed architecture, presented in the Figure 3.3, we diverged from the utilization of XSS and SQLi vulnerabilities. Instead, we opted for Java as the target programming language and chose to focus on detecting weaknesses such as NULL Pointer Deference and Command Injection. This work proposes a tool based on a static analysis and machine learning for finding vulnerabilities caused by unchecked input.

The primary contribution of this research comprises the development of a prototype capable of identifying vulnerable Java source code through the application of a probabilistic classifier and features derived from static code analysis. To achieve this, we will compile a dataset for training and testing our classifiers. This dataset will be sourced from the National Vulnerability Database (NVD) [64] and the Software Assurance Metrics And Tool Evaluation (SAMATE) project [76].

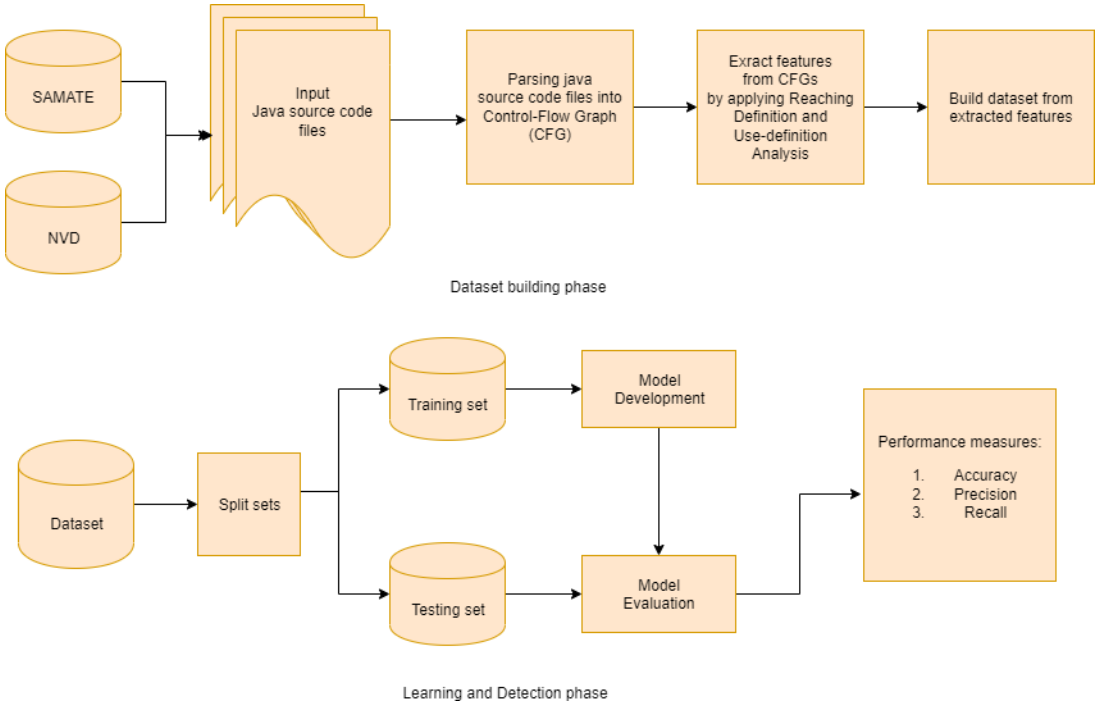



Figure 3.3: A high-level flow of proposed architecture

As depicted in Figure 3.3, the project will be organized into two primary phases. The initial phase comprises dataset building, wherein we will generate Control Flow

Graphs (CFGs) from Java source files and extract corresponding features. Subsequently, the following phase involves learning and detection. During this phase, we will divide the dataset into two subsets: the training set, designated for model training, and the testing set, utilized for model assessment.

Similarly to the approach undertaken by Kronjee et al [51], we intend to employ the Abstract Syntax Tree (AST) and control-flow graph as the foundational sources for extracting features. We are of the opinion that the AST could provide substantial information about the code, which could be highly valuable, including discerning whether a token represents a function or a variable, as well as identifying the functions applied to specific variables.

Reaching definition and taint analysis will be used to identify potential vulnerabilities. Utilizing the AST, we can determine that a variable is being used as a parameter for a call to function. This means that we can provide these functions and their respective argument values (variables) as features to our probabilistic classifier. By applying this method, our machine learning algorithm will learn which function is likely to be a potentially vulnerable function given enough samples. To determine if a function is applied to a variable, we can construct a CFG, from the AST. By considering the entire preceding execution path, we can use the invocation of functions in that path as a feature.

When applying taint analysis, the AST can provide us information if a variable has been assigned using a literal value, like a number or a string. In that case, the data is from a trusted source and the variable is untainted. However, if the variable's value is derived from a source like another variable or a function—especially if this source isn't a literal—the variable must be treated as tainted. Subsequently, anything that is set using this variable must be also considered tainted as well. By using this approach we can determine whether a variable contains data from a specific source and whether it's tainted or untainted.

To facilitate feature extraction, we require a parser capable of translating code into an AST, subsequently enabling the construction of a CFG. To accomplish this, we will utilize the "spoon-control-flow" component [66]. This module offers the capability to generate both control-flow and data-flow graphs for a Java program, relying on its AST that is also created using the Spoon library. Subsequently, we will implement reaching definitions analysis and use-definition techniques to extract features from the AST.

To programmatically determine the reaching definitions from our AST, we used the algorithm described in Section 2.5.1, based on the description from [3]. The referenced book, primarily deals with code blocks. However, in our specific scenario, we have

modified the implementation. This adjustment was necessary due to complexities in implementation, leading us to focus on control-flow nodes rather than traditional code blocks.

After building the Reaching Definitions sets, our next step involves establishing the Use-Definition (UD) chains for each definition. UD chains comprise a utilization of a variable U, and all the definitions D of that variable that can reach that use without any other intervening definitions in between.

For each node within the CFG of the program, our objective is to determine which functions may have been used in the paths to that nodes. We consider the invocation of functions as features, hence, in our feature set, we use UD chains to create new features. We consider each node to be a sample that can potentially be categorized as either vulnerable or not vulnerable.

As explained in Section 2.5.2, in order to detect vulnerabilities, there needs to be a potentially vulnerable function and a lack of sanitization, which usually comes in the form of a function as well.By utilizing the presence of functions as features, we can derive a feature set wherein all functions invoked using values from potentially untainted variables serve as our features.

# 4

# Implementation

In the this section, the technical aspects of the proposed architecture and some challenges, and obstacles found during the implementation are described in more details.

## 4.1 Data Collection

Before implementing the tool, the first step was to find a large amount of java projects related to our vulnerabilities case study (Null Pointer Deference and Command Injection). Since the goal is to cover those vulnerabilities, many examples for each of those vulnerability types are required.

The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project maintains several tests cases for several programming languages and various vulnerabilities types. It is based upon the Common Vulnerabilities and Exposures (CVE) [15], standard vulnerability dictionary. The vulnerabilities contained in the SAMATE are classified according to the Common Weakness Enumeration (CWE) [16], including Java test cases for command injection (CI) and Null pointer deference (NPD) vulnerabilities. It was possible to filter and derive 800 projects for CI and 600 projects for NPD. Each project, was downloaded from SAMATE to a local machine for processing.

The National Vulnerability Database (NVD) does not maintain vulnerable source code projects, just the links with information about vulnerabilities reported. And it doesn't maintain information organized like SAMATE does. In order to get data from it, it

would required some extra time, such as create our own list of Java projects using various sources. The process would involve searching for sources and attempting to identify projects with vulnerabilities for our case study. Subsequently, these projects would be organized manually based on our tool's input. For this reasons we were not able to use NVD and we consider this task as our future work.

## 4.2 Data Transformation

This section corresponds to the first phase in the proposed architecture (see Figure 3.3). Describes all the implementation done about this part of architecture.

In order to process and create the dataset from SAMATE data, each java project has to be transformed to AST using the spoon library. Then all the elements of the AST corresponding to the functions are filtered. Next, each function is represented in CFG in order to apply data-flow analysis. Afterwards, Reaching Definition is determined, and then the Use-Definition chain. Figure 4.1 illustrates the transformation process.

From the use-definition chain, it was possible to determine which are the possible tainted variables (in the case of CI) or possible variables with null values (in the case of NPD) that are being used in invocation of the potentially vulnerable methods in each function. These methods, and values of the arguments are extracted as features for the datatset, where invoked methods correspond to the feature **Func** and the values of the arguments passed to these methods correspond to the feature **Var** as illustrated in the Figure 4.2.

To identify vulnerable rows in dataset (model output), we used XML files (manifest.sarif) that contain informations about project, including the name of the java files and the number of vulnerable rows. This XML file exists for each project, And so, we were able to assemble our dataset by including all of the vulnerable code snippets from SAMATE projects.

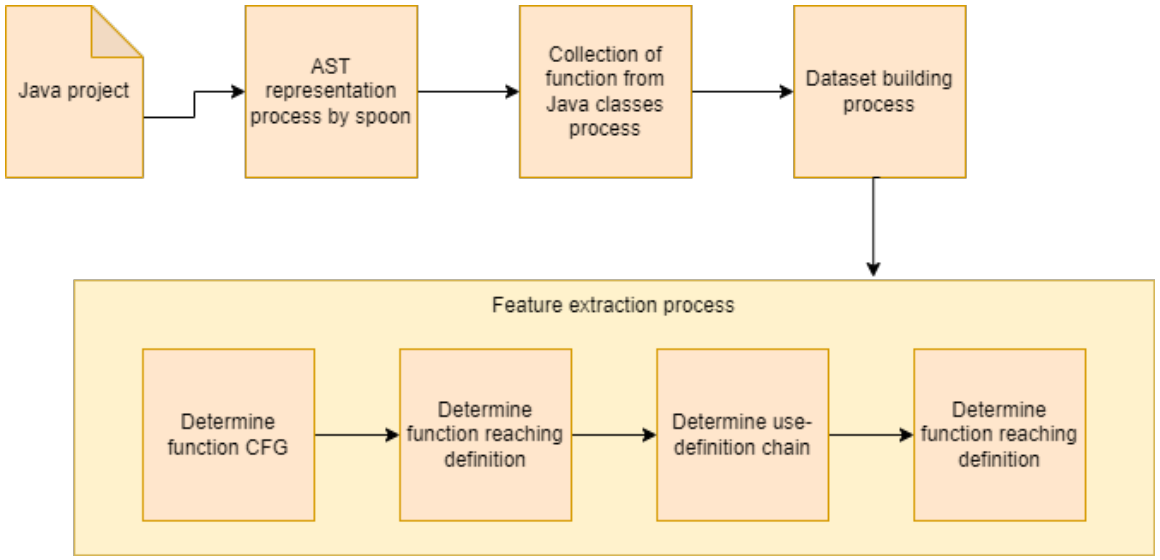



Figure 4.1: Java project transformation process

To create the dataset, there are dependencies between different processes, as shown in the Figure 4.1. The representation of the source codes was transformed into AST and then CFG to facilitate static analysis of the source code. With the use-definition chain, it is possible to determine which variables are being used in the invocation of each function. These functions and their respective argument values are extracted to build the dataset. Figure 4.2 represents the description of the dataset.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| | | Features | | Classe | |
| | | **Func** | **Var** | **Vuln** | |
| 1 | | File() | "cmd.exe format c:" | 0 | |
| 2 | | File() | readLine() | 0 | |
| 3 | | Exec() | "cmd.exe format c:" | 1 | |
| 4 | | Exec() | readLine() | 1 | |
| 5 | | Exec() | "cmd dir" | 0 | |
| 6 | | println() | | | |
| 7 | | ProcessBuilder() | | | |

Figure 4.2: Dataset description

The attribute "Func" represents potentially vulnerable invoked methods. The attribute "Var" represents the values of the arguments passed to the methods. The attribute "Vuln" represents the class of the dataset. It indicates whether a row is vulnerable or

not.

The attributes "Func" and "Var" represent characteristics of our problem, serving as inputs to the model. The attribute "Vuln" represents the model's output, indicating whether a row is vulnerable or not.

The model learns whether the functions are vulnerable or not based on the values of the arguments that are passed to these functions. The dataset has been saved in a Comma-separated values (CSV) file.

## 4.3 Model Creation

This section corresponds to the second phase of the proposed architecture.

Once we have the dataset created and ready to be used in training, we are able to create the model. This section describes the entire implementation process.

Figure 4.3 illustrates the actions carried out in the model creation process. Next, we will describe these steps.

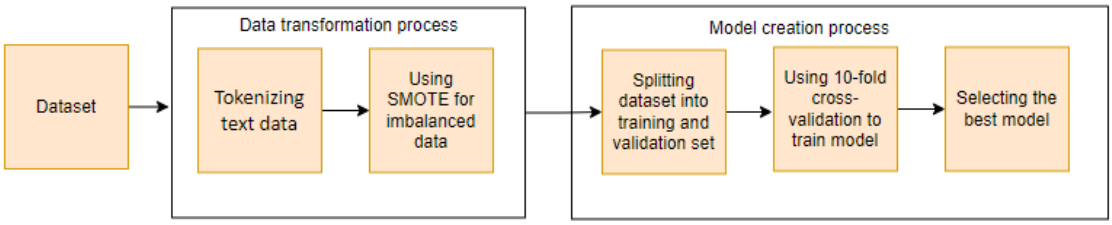



Figure 4.3: Overview of the methodology used to create the model.

1. **Tokenizing text data**

   In order to perform machine learning on text data, we first need to transform text content into numerical feature vectors [85]. Since we are using text data, it requires special preparation before we can start using it for predictive modeling.

   The text must be parsed to isolate words, in a process called *tokenization*. Then the words need to be encoded as integers or floating point values for use as input to a machine learning algorithm, a process known as *vectorization*.

   This can be done by assigning to each word a unique number. Any text in our data can be encoded as a fixed-length vector with the length of the vocabulary

of known words. The value in each position in the vector could be filled with a count or frequency of each word in the encoded document.

This is the bag of words model [85], where we are only concerned with encoding schemes that represent what words are present or the degree to which they are present in encoded documents without any information about order.

We used the scikit-learn library tools to perform tokenization of text data [79].

2. **Using Synthetic Minority Oversampling Technique (SMOTE) for imbalanced classification data**

   Given that in our dataset the distribution of classes is significantly skewed, the class "not vulnerable" (the minority class) has a much smaller number of samples compared to the "vulnerable" class (the majority class) [82].

   Our model's training data has been biased towards the majority class, making it challenging for the model to accurately predict the minority class. As a result, the classification algorithms have been biased in favor of the majority class, leading to poor performance for the minority class.

   To solve this problem, we were forced to oversample the examples in the minority class (vulnerable class). This have been achieved by simply duplicating examples from the minority class in the dataset prior to fitting a model. This was able to balance the class distribution without providing any additional information to the model.

   The improvement on duplicating examples from the minority class is to synthesize new examples from the minority class. This is a type of data augmentation for tabular data and can be very effective.

   We used the most common and widely approach to synthesizing new examples called the Synthetic Minority Oversampling Technique (SMOTE). This technique is described by Nitesh Chawla, et al. in [11].

   SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line [82].

   In our python project, we used the implementations provided by the imbalanced-learn Python library [38], which we installed via pip.

3. **Splitting dataset into training and testing set**

   After parsing our text data into numerical feature vectors and then applying

SMOTE to balance our class, the next step was to split the dataset into the following subsets:

**Training Set** was used to train the machine learning model. The model learns from the patterns and features present in the training data to make predictions on new, unseen data.

**Testing Set** was used to evaluate the performance of the trained model. It serves as a proxy for new, unseen data and helps assess how well the model generalizes to unseen examples.

The purpose of splitting the dataset was to avoid *overfitting*, where the model performs well on the data it was trained on but poorly on new data. By having a separate testing set, you can assess the model's performance on unseen data and choose the best parameters for the model to achieve better generalization.

The dataset was randomly divided into the training and testing sets, with the split ratio 60% for training and 40% for testing.

4. **Using K-fold cross-validation**

Is a common technique used to assess the performance of a model. It involves dividing the dataset into k equal parts (folds) and then training and evaluating the model k times. In each iteration, one fold is used as the test set, and the remaining nine folds are used as the training set. This process is repeated ten times, ensuring that each fold is used as the test set once. The results of the ten evaluations are then averaged to obtain a more robust and reliable performance estimate for the model [41].

The main reason for using k-fold cross-validation was because it is easy to understand, implement, and generally results in less biased estimates (less optimistic estimation of the model) compared to other methods. It also allows the random division of data.

Sampling bias is a systematic error due to a non-random sample of a population, causing some members of the population to be less likely to be included than others, resulting in a biased sample [41].

The procedure has a parameter called k, which refers to the number of groups to which each data sample should be divided. Then, the average of the result is calculated for each instance (group).

The procedure is as follows:

- Randomly divide the data into K-folds (K groups). The higher the value of K, the less biased the model.
- Fit the model using the first K-1 groups of data and validate the model using the remaining K group of data.
- Repeat the procedure until each K-fold (each group) serves as the test dataset. Then, take the resulting average of each instance. This is the performance metric for the model.

There are 2 variations of cross-validation commonly used: Stratified and Repeated, both available in scikit-learn [75].

**Stratified**: The division of data into groups can be more strict. It may have criteria such as ensuring that each group has the same proportion of observations with a certain categorical value, such as the class outcome value. Meaning that each group or data division will have the same distribution of examples per class existing in the entire training dataset.

**Repeated**: The validation procedure is repeated n times. The crucial aspect is that the data sample is shuffled before each repetition, resulting in a different split of the sample.

We had used stratified 10-fold cross-validation to estimate model accuracy. This had split our dataset into 10 parts, trained on 9, and tested on 1, and repeated for all combinations of train-test splits. We used the metric of 'accuracy' to evaluate models. This is a ratio of the number of correctly predicted instances divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g., 95% accurate). We used the scoring variable to build and evaluate each model.

5. **Build multiple different models to predict vulnerability from our dataset**

   Since we didn't know which algorithms would be good for our problem or what configurations to use, we had to create different models using various algorithms.

   We used a mixture of simple linear, with nonlinear algorithms. We decided to also use deep learning (Multi-layer Perceptron classifier) to make a small experiment. The followings algorithms were used on the process:

   - Logistic Regression (LR)
   - Decision Tree (DT)
   - Neighbor Classifier (NC)
   - Naive Bayes (NB)
   - Support Vector Machine (SVM)
   - Multi-layer Perceptron classifier (MLPC) - Deep Learning.

6. **Select the best model as our final model**

   After building and evaluating different models, we compared the models to each other and selected the most accurate one as our final model.

   Afterward, we used the testing set to understand the accuracy of the final model. We fitted the final model on the entire training dataset and made predictions on the testing dataset. We evaluated the predictions by comparing them to the expected results in the testing set, then we calculated the classification accuracy, as well as a confusion matrix and a classification report. All the experimental results are described in the Chapter 5.

# 5

# Evaluation

The models were trained on the training datasets and their performance was evaluated with the testing dataset. This section describes experimental results obtained during the evaluation with testing set and other data from open-source projects. The chapter is also dedicated to discussing the results and comparing them to other related works in the field.

## 5.1 Model Performance

A plot of the model evaluation was created, where the spread and the mean accuracy results of each model were compared. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (via 10 fold-cross validation).

The data for the Command injection vulnerability was split into a 60% training set and a 40% test set, resulting in 7,872 samples for training and 5,248 for testing. The split was performed using the stratify method, which ensures that the proportion of target classes is preserved in both the training and testing sets. For the Null Pointer Dereference vulnerability, the data splitting process resulted in 1046 samples for training and 764 samples for testing.

Given our uncertainty about which algorithms and configurations would be suitable for our problem, we have created different models using the algorithms we previously mentioned.

**The training set** was used to train the selected models. It was where our models learns patterns and relationships within the data, and it served as the basis to choose the final model for the predictions.

During training, the model was exposed to the features (input data) and their corresponding target values (labels or output) from the training set. Model learning is carried out by adjusting its internal parameters based on the training data.

**The testing set** was used to evaluate the final model's performance on unseen data. It provided an estimate of how well our final model was likely to perform on new, real-world data.

Once the model was trained, it was evaluated on the testing set. The model's predictions were compared to the true target values, and performance metrics (e.g., accuracy, precision, recall) were calculated. This helped us in understanding how well the model generalized to data it hadn't seen before as we'll see next.

Before splitting the data, the SMOTE technique was applied, since we were having problem of class imbalance in dataset. This was done to ensure that the classes had approximately the same number of instances. This technique helped our model better learn and generalize from the data, resulting in improved classification performance.

**Note:** All tests, experiments, and evaluations presented in this section were conducted using the SAMATE data for the reasons explained earlier (Section 4.1). Despite the fact that we have used only SAMATE data, we were able to ensure that the results can be directly compared and provide a stable baseline for our analysis. The utilization of SAMATE data, with a sufficient number of examples, simplifies the assessment of the model's performance and enables clear comparisons between different configurations and techniques.

Table 5.1: Statistical summary of average performance among different probabilistic classifiers for Null Pointer Dereference.

| **Vulnerability Type** | **Null Pointer Deference** | |
|---|---|---|
| Classification method | mean accuracy | Standard deviation Accuracy |
| Logistic regression | 0.996154 | 0.007692 |
| Decision tree | 0.923385 | 0.024098 |
| Support vector machines | 0.996154 | 0.007692 |
| Naive Bayes | 0.996154 | 0.007692 |
| Neighbor classifier | 0.996154 | 0.007692 |
| Multi-layer Perceptron classifier | 0.999846 | 0.000462 |

Table 5.2: Statistical summary of average performance among different probabilistic classifiers for Command Injection.

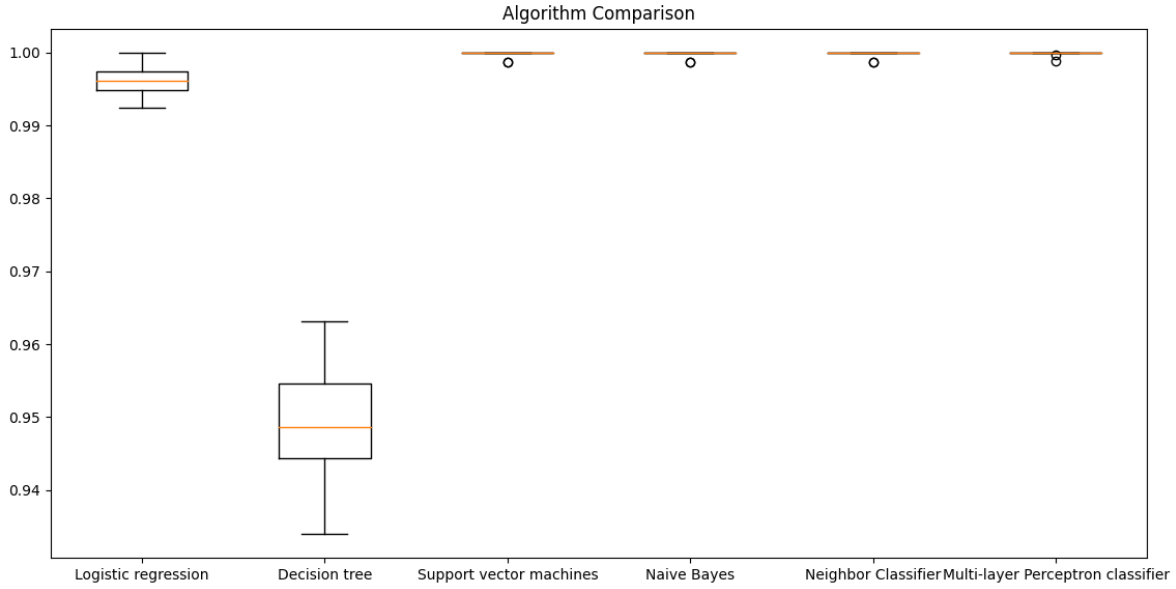
| **Vulnerability Type** | **Command Injection** | |
|---|---|---|
| Classification method | Mean Accuracy | Standard deviation Accuracy |
| Logistic regression | 0.998096 | 0.001528 |
| Decision tree | 0.94919 | 0.008034 |
| Support vector machines | 0.999873 | 0.000381 |
| Naive Bayes | 0.999873 | 0.000381 |
| Neighbor classifier | 0.999873 | 0.000381 |
| Multi-layer Perceptron classifier | 0.999772 | 0.000684 |

Tables 5.1 and 5.2 show the results of the mean and standard deviation accuracy of the several probabilistic classifiers for Null Pointer Deference and Command Injection vulnerabilities.

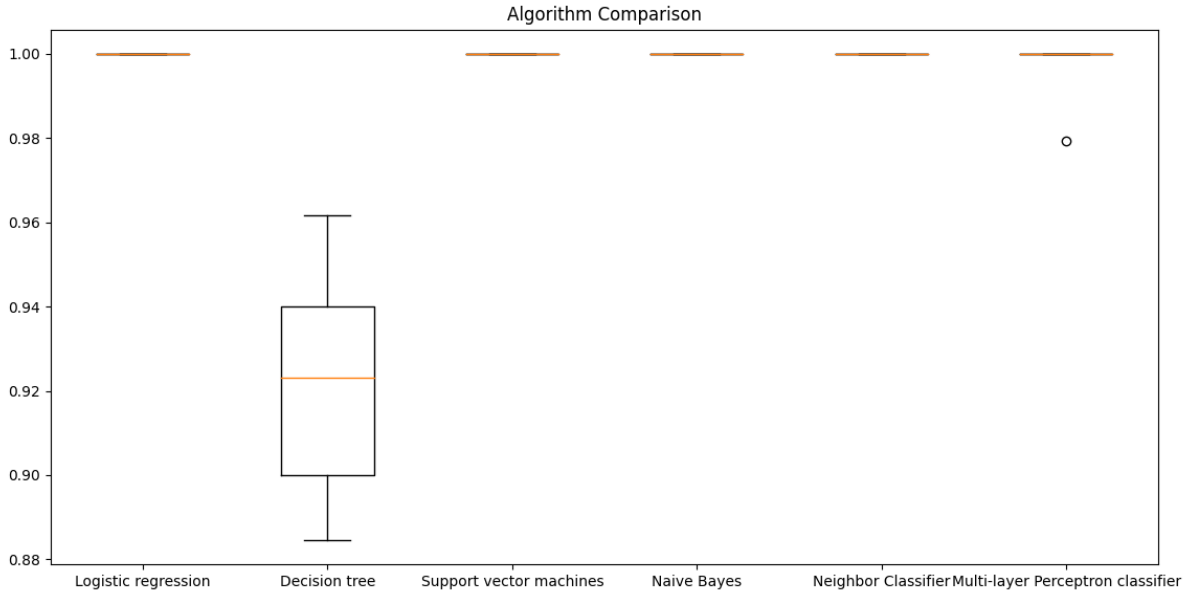
Figures 5.1a and 5.1b show the box and whisker plots for each model, using the accuracy measure. Therefore, LR, NB, SVM, NC, and MLPC are the best algorithms for our dataset, as Figure 5.1 illustrates. There are more options in choosing algorithms, which is beneficial for our model.

We can see in Figure 5.1 that the box and whisker plots are squashed at the top of the range, with many evaluations achieving almost 100% accuracy, and some pushing down into the high 90% accuracy (in case of Decision Tree).

Given that there are several algorithms with good accuracy, we choose Logistic Regression to use in our final model. Next, the evaluation results obtained with the test set are presented for each type of vulnerability.




(a) CI vulnerability




(b) NPD vulnerability

Figure 5.1: Box and Whisker Plot Comparing Machine Learning Algorithms for NPD and CI vulnerability

Figures 5.2a and 5.2b show the confusion matrix results for the LR models for both vulnerabilities, illustrating the true and predicted classifications for our test set.

**True Positive**: These are instances that were correctly predicted as vulnerable by the models. The models correctly identified 282 instances for NPD and 2624 for CI as vulnerable.

**False Positive**: These are instances that were incorrectly predicted as vulnerable by the model. The model incorrectly identified 0 instances for NPD and 0 for CI as vulnerable which is advantageous in our case.

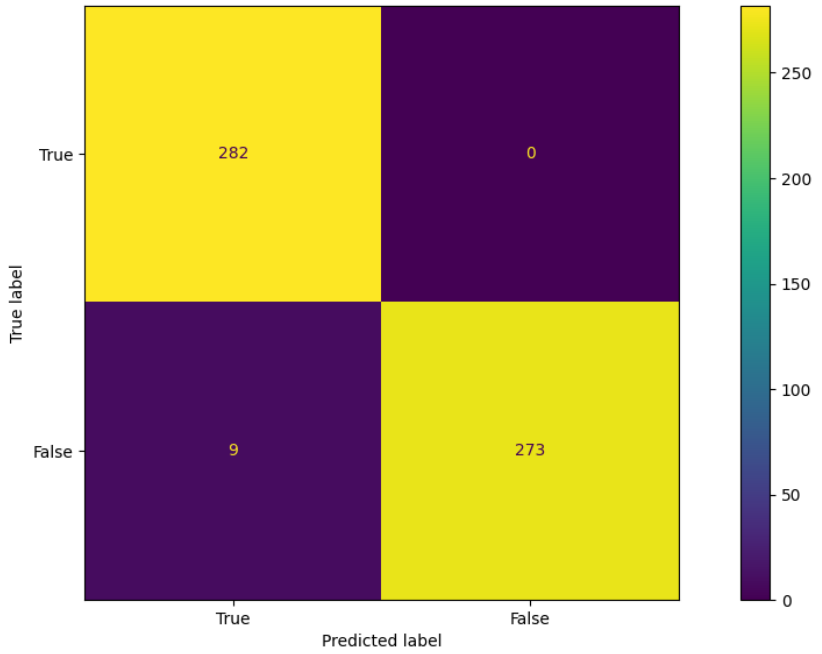
**True Negative**: These are instances that were correctly predicted as not vulnerable by the model. The model correctly identified 273 instances for NPD and 2508 for CI as not vulnerable.

**False Negative**: These are instances that were incorrectly predicted as not vulnerable by the model. The model missed 9 instances for NPD and 116 for CI that were actually vulnerable.

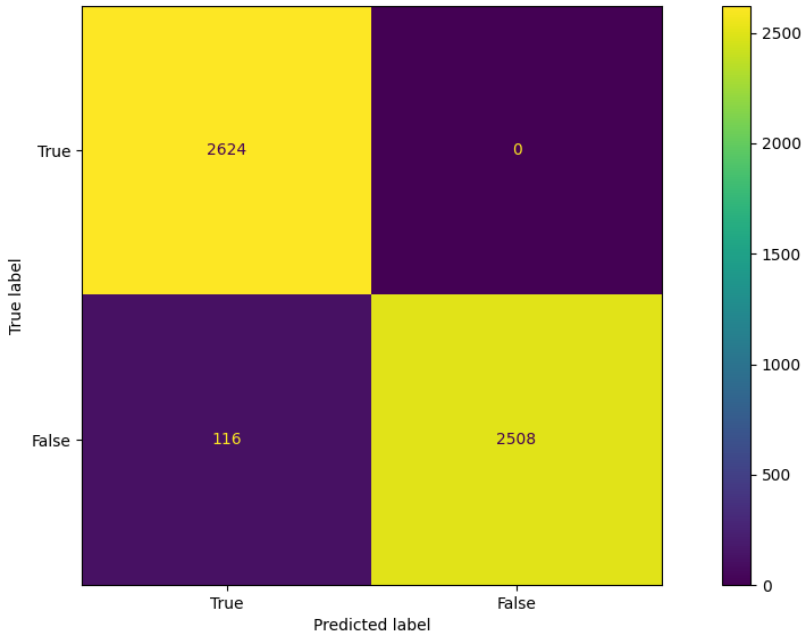
The precision-recall curve plot is represented in the Figures 5.3a and 5.3b showing the precision/recall for each threshold for the logistic regression model. The graph represents recall on the x-axis and precision on the y-axis.

Once we have an Area Precision of 0.99, closer to 1, this means that the model has a good balance between precision and recall, effectively classifying well instances despite of false positives cases.

Figure 5.4 illustrates different classification metrics provided by scikit-learn tool's classification reports, detailing how well the model is performing on different classes.

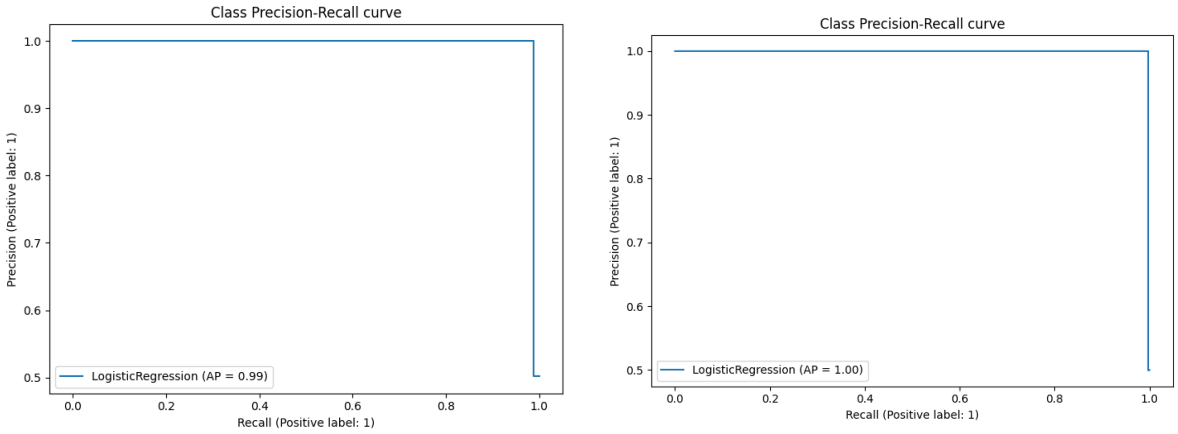



(a) NPD vulnerability




(b) CI vulnerability

Figure 5.2: Comparing LR model confusion matrix for NPD and CI vulnerability




(a) NPD vulnerability

(b) CI vulnerability

Figure 5.3: LR model Precision-Recall Plot for NPD and CI vulnerability

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.91 | 1.00 | 0.95 | 170 |
| 1 | 1.00 | 0.90 | 0.95 | 171 |
| accuracy | | | 0.95 | 341 |
| macro avg | 0.95 | 0.95 | 0.95 | 341 |
| weighted avg | 0.95 | 0.95 | 0.95 | 341 |

(a) NPD vulnerability

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.96 | 1.00 | 0.98 | 2624 |
| 1 | 1.00 | 0.95 | 0.98 | 2624 |
| accuracy | | | 0.98 | 5248 |
| macro avg | 0.98 | 0.98 | 0.98 | 5248 |
| weighted avg | 0.98 | 0.98 | 0.98 | 5248 |

(b) CI vulnerability

Figure 5.4: Comparing LR classification report for NPD and CI vulnerability

Figures 5.4a and 5.4b show the classification report for different metrics such as precision, recall, F1-score, and support. The results show a high rate of these metrics for both vulnerable and not-vulnerable instances despite of false positive cases.

### 5.1.1 Discussion

In this scenario, since a part of SAMATE data was also used to test the final model, we obtained good results. The models performed well for both vulnerabilities. There were a few cases of false negatives where the model failed to identify some functions that are actually vulnerable.

We also have a good number of false positives, indicating that the model successfully identified all non-vulnerable functions. However, we will later discover that this number is just a false impression in the testing scenario with data from other projects. This is due to the fact that there are some functions that are supposedly not vulnerable but receive arguments with patterns identical to vulnerable functions.

It is necessary to further fine-tune the 'Var' attribute in order to match the pattern exactly (the same string if possible) with the truly vulnerable codes. This is because each project specifies values differently, which ended up being different from how it was specified in the SAMATE projects.

We have come to the conclusion that we have good models for both types of vulnerabilities with good performance, even with the cases of false positives, it was able to detect vulnerabilities in truly vulnerable places.

## 5.2 Model Evaluation with data from other projects

Given the good results with SAMATE data, we decided to analyse if our tool could detect vulnerabilities from real software projects.

We searched for some examples of vulnerable Java code on GitHub and found the following projects for Command injection vulnerability: Vulnerable Java Application [25], Java Sec Code [29], Operation System Command Injection Vulnerability in Java Spring [32], Amaze File Manager [31], OpenTSDB 2.4.0 Remote Code Execution [24], OS Command Injection [30], and the following projects for Null Pointer Deference vulnerability: null-pointer-dereference-examples [28], Selenium [27] and dbus-java [26].

For each project, we transformed the source code into dataset (feature extraction) with our dataset generator tool, by passing the project directory path to the tool, and then used the models trained with SAMATE data to find vulnerabilities. The output of our tool was a CSV file containing the file name, line number, node number and probability for each instance. We used manual inspection to confirm whether or not the projects are vulnerable.

The results are resumed in Tables 5.3 and 5.4. The "Project" field indicates the name of the project. The "Vuln. Lines" indicates the number of vulnerable lines predicted by the model. The "Non-Vul. Lines" indicates the number of non-vulnerable lines predicted by the model. The "Tot. lines" indicates the total number of lines extracted from the project. The "Comments" provides a qualitative assessment of the model's behavior in that project.

This qualitative assessment provides insights into the model's stability and generalization across different projects, even though we don't have ground truth labels for specific evaluation metrics.

Table 5.3: Result of model evaluation using real software projects for Null Pointer Deference vulnerability.

| **Project** | **Vuln. Lines** | **Non-Vul Lines** | **Tot. lines** | **Comments** |
|---|---|---|---|---|
| null-pointer-dereference | 25 | 160 | 185 | The model was able to detect the vulnerability, but the prediction was inconsistent. |
| dbus-java | 38 | 238 | 276 | Incorrectly predicted. the prediction was inconsistent. |
| SeleniumHQ | 70 | 1149 | 1219 | The model was able to detect vulnerable lines, but the prediction was inconsistent. |

Table 5.4: Result of model evaluation using real software projects for command injection vulnerability.

| **Project** | **Vuln. Lines** | **Non-Vul Lines** | **Tot. lines** | **Comments** |
|---|---|---|---|---|
| Vulnerable Java Application | 1 | 5 | 6 | Predicted correctly. The prediction was consistent. |
| Java Sec Code | 20 | 218 | 238 | Incorrectly predicted. The prediction was inconsistent. |
| OS Command Injection Vulnerability in Java Spring | 0 | 1 | 1 | In this project, it was only possible to extract one line to create the dataset, which is actually vulnerable. Unfortunately, the model missed it. The prediction was inconsistent. |
| ArmazeFileManager | 116 | 1090 | 1206 | The model was able to detect the vulnerable functions. The prediction was inconsistent. |
| OpenTSDB 2.4.0 Remote Code Execution | 220 | 2085 | 2305 | The model was able to detect the vulnerability, but the prediction was inconsistent. |
| OS Command Injection | 0 | 3 | 3 | Incorrectly predicted. The prediction was inconsistent. |

As shown in the tables 5.4 and 5.3 and explained earlier, the model is also detecting vulnerabilities (false positive cases) in functions that even though they receive variables whose values may come from input or null, and therefore tainted, are not actually vulnerable. Some of these functions don't even receive arguments, therefore, it doesn't make sense for them to be vulnerable.

### 5.2.1 Discussion

Based on this analysis, we can observe that in a significant number of projects, the models (for both type of vulnerabilities) predictions were inconsistent, meaning that they tend to be less stable (predicting some instances incorrectly). Nevertheless, it's worth noting that the models were successful in detecting vulnerabilities in certain projects.

We believe that the 'Var' feature requires further refinement to precisely match the pattern with the actual vulnerable code, ideally matching the same string. This is necessary because, as previously explained, each project specifies values in a different format than the SAMATE projects. Specifically, each project defines its own path for command execution, even when the commands themselves are identical. In other words, we think that the value of the 'Var' feature in the SAMATE dataset is different from the actual codes (due to each project being distinct), which is influencing significantly the vulnerability detection.

For CI vulnerability: The source code contains file names and paths used in command execution, and these file-related details are considered as part of the model's input for detecting CI vulnerabilities.

For NPD vulnerability: the source code contains functions that receive variables as arguments. However, it's challenging to determine whether these variables are genuinely null through the static code analysis, especially when the variable serves as a parameter of a function. In other words, static code analysis cannot provide a straightforward way to identify null variables in the context of NPD vulnerabilities.

The structure of the 'var' attribute needs further refinement to ensure it is most suitable for the model, enabling the model to learn and make predictions effectively. Due to time constraints, this couldn't be done, and therefore, it remains for future work.

## 5.3 Comparison with other works

This section consists of comparisons with similar works in the field to provide a perspective from which observations and measurements are made for the evaluation of this work. As there are fundamental differences in each approach, direct comparison cannot be drawn between the outcomes. The results are summarized in Table 5.5.

The current design of our tool is limited to Java programs. Future research should be conducted to adapt it for use with other programming languages.

The approaches are compared under the following aspects:

- **Scope and applicability**: has the model been trained on a single project and can it only classify files within that application, or is it generally applicable to any code from a large variety of sources;
- **Programming language**: what programming language was subject of the study;
- **Machine Learning Approach**: what machine learning approach was used;
- **Vulnerability types**: which kinds of vulnerabilities can be detected;
- **Dataset**: Is the data sourced from real-world projects or from synthetic databases, such as SAMATE data.

Table 5.5: Results of comparisons with similar works

| **Aspects of the approach** | | | | | |
|---|---|---|---|---|---|
| **Tool Name** | **Prog. Lang.** | **Vul. Type** | **Scope & applicability** | **Mach. Learn. Approach** | **Dataset** |
| SWD-SCA-ML (our tool) | Java | CI, NPD | General | Supv. Learn., Prob. Classifiers | Real & Synth. |
| WIRECAML | PHP | SQLi, XSS | Inconclusive | Supv. Learn., Prob. Classifiers | Real & Synth. |
| VUDENC | Python | SQLi, XSS, CI, XSRF | General | Supv. Learn., Deep Learn. | Real & Synth. |
| VulDeePecker | C/C++ | resource management error, buffer error | General | Supv. Learn., Deep Learn. | Real & Synth. |

### 5.3.1 Discovering vulnerabilities using data-flow analysis and machine learning

This is the main work related to our problem, proposed by Kronjee et al. [51]. It combines static code analysis with machine learning for detecting SQL injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities in PHP applications. It also uses SAMATE for collecting training data for the model [51].

The work attempted to contribute and demonstrate that it is possible to combine machine learning with static code analysis for detecting vulnerabilities in software. However, we suggest that the demonstrated results show some uncertain, especially when testing the tool with different data (data from other applications). This is because they did not demonstrate the data extracted from other applications. In other words, their work did not include a demonstration of the data extracted from other applications, which was used to test their tool. Furthermore, based on our analyses, we suggest that their process for feature extraction and dataset creation is based on a method that does not consistently define features according to the function names and variable number of features. The features extracted from the source codes are functions (function names), which can differ from project to project, and the size of features could also vary from project to project. In this sense, the work may not be considered definitive, as it could be challenging to apply the model created and trained with a specific dataset (e.g., SAMATE data) to any other dataset extracted from a different project or source code.

The authors built a tool called WIRECAML [51] and Table 5.6 presents the results of WIRECAML using various classifiers trained on SAMATE and NVD data. We have made comparisons regarding the accuracy of their model with ours for some classifiers that we have used in common, since it's not possible to make a comparison regarding the types of vulnerabilities used in both cases.

The main advantage of our tool over theirs is its versatility. In our case, our model can be applied to any dataset generated using our tool. On the other hand, in their case, it remains inconclusive whether their model can be applied to any dataset generated by their tool, due to the features that they have considered as previously explained in the preceding paragraph.

Our model achieved better results when using the same classifiers compared to their model. Their work has limitations in feature extraction for dataset construction, and this represents the primary advantage of our work over theirs. The model created by our tool can be applied to datasets extracted from other applications since it considers the same features.

### 5.3.2 VUDENC - Vulnerability Detection with Deep Learning on a Natural Codebase

It is a vulnerability detection system that utilizes neural networks to learn vulnerable features from a collection of data extracted from GitHub. The work focuses on the

Table 5.6: Comparison of several probabilistic classifiers in terms of AUC-PR values extracted from [51]

| Classification method | Vulnerability | AUC-PR |
|---|---|---|
| Decision tree | SQLi | 0.88 |
| Random forest | SQLi | 0.85 |
| Logistic regression | SQLi | 0.87 |
| Naive Bayes | SQLi | 0.64 |
| TAN | SQLi | 0.75 |
| Decision tree | XSS | 0.82 |
| Random forest | XSS | 0.82 |
| Logistic regression | XSS | 0.79 |
| Naive Bayes | XSS | 0.69 |
| TAN | XSS | 0.81 |

Python programming language and exclusively learns source code [89].

This work managed to demonstrate the potential use of deep learning directly on source code to learn vulnerable features, utilizing LSTM (Long Short-term Memory) models [6].

They created the model using LSTM and managed to achieve good results. With an F1 score of around 87%, they attained a precision of 91% and a recall of 83%.

Their tool was able to detect several types of vulnerabilities, and this is the main advantage of their work compared to ours. Another advantage in comparison with our approach is that they used real GitHub data for model creation. In our case, we used SAMATE data for training and model creation.

### 5.3.3 VulDeePecker Deep Learning Based System For Vulnerability Detection

Li et al. [95] developed this tool to detect buffer error vulnerabilities and resource management error vulnerabilities in C/C++ programs. They work on a "code gadget database" made from a large number of popular open source projects, including the Linux kernel and Firefox. The vulnerabilities are found by using the NVD and SAMATE dataset, which contain synthetic and real-life / production code, flaws, and vulnerabilities.

Selecting their data from those sources, Li et al work on very high-quality code. Similar to our approach, they also collected projects with vulnerable source code to extract their own features and create the dataset. They labeled all vulnerable locations in

source codes, just as we did, but they did so manually. This differs from our approach, which automates this entire process.

They train a bidirectional LSTM and achieve an impressive F1 score of around 85-95%. Our tool achieve quite the same high score too. Furthermore, their database only consists of synthetic code as well.

The authors of VulDeePecker compared their tool to other approaches and found a precision of 25% for the Flawfinder tool [23], 39.6% for Checkmarx [12], and 19.4% for RATS [72], in contrast to VulDeePecker's precision of 88.1%. Which means that VulDeePecker is more effective than these tools. The VulDeePecker approach successfully was able to reduce the number of false positive results to almost zero, a level of achievement that we were unable to accomplish in our case.

## 5.4 Limitations

The present design, implementation, and evaluation of this work have several limitations, which suggest interesting open problems for future research. First, the present design of this work is limited to deal with a variety of different types of vulnerabilities. The feature extraction process is a challenging problem for projects compatibilities with Java programming language version 8, as well as with higher versions due to the functional programming paradigms (dealing with lambda expressions on features extraction).

Second, the present design of our tool only deals with Java programs. Future research needs to be conducted to adapt it to deal with other kinds of programming languages.

Third, the present design of this work only deals with vulnerabilities related to functions invocation. The future work could investigate how to detect the other kinds of vulnerabilities by leveraging the other kinds of key points.

Fourth, The data collected for model creation was just from one source (SAMATE) and insufficient, it's necessary to collect more data from different sources and from real example source code.

Fifth, the evaluation of the model's performance was conducted using standard measures commonly used in the context of vulnerability prediction, such as predictions, accuracy, precision, and recall. By using these well-established metrics, the study aimed to minimize potential risks to the validity of its conclusions. However, in practice, there may be other metrics and representation demonstrating how well a classifier performs. In essence, the evaluation process used common and accepted measures but

we acknowledge the possibility of further metrics that might be valuable for assessing classifier performance.

Sixth, our current tool implementation is limited to the use of machine learning classifiers. To enhance our systematic experiments, we can explore alternative learning techniques for vulnerability detection, including Natural Language Processing, Recurrent Neural Networks, and Convolutional Neural Networks (CNNs). It's also possible to combine these methods for a more comprehensive and holistic approach.

Seventh, the present method of model creation is limited because the hyperparameter tuning process was not used. It can be a crucial step in model creation to tune parameters in machine learning in order to adjusting the settings of a machine learning algorithm or model to ensure that the model performs at its best.

Finally, furthermore, design decisions were made which might affect the overall end result.

# 6

# Conclusion

In this work, we have presented Software Weakness Detection, a system for vulnerability detection based on Machine Learning applied to source code. The research purpose is to relieve human experts from the time-consuming and subjective work of manually defining features for vulnerability detection. This work demonstrates the potential of using machine learning directly on source code in order to learn such vulnerability features by leveraging classifier models.

The system works on Java source code and can predict 2 different types of vulnerabilities (Command Injection and Null Pointer Deferences). To create the basis for the tool, a large dataset was derived from SAMATE projects and transformed according to the model inputs. The data was transformed from several SAMATE projects examples containing code snippets and for 2 different types of vulnerabilities.

A classifier model based on Logistic Regression algorithm has been trained on a large data of java source code to be able to perform vulnerabilities detection. Systematic experiments show that our tool achieves, on average, an accuracy of 96.5%, a recall of 95%, a precision of 95% and an F1 score of 96.5% using SAMATE data with a ratio of 60% for the training dataset and 40% for the test dataset. Similarly, with data from real projects, we achieved favorable results. Despite encountering some false positives, the model was able to detect vulnerabilities in genuinely vulnerable locations. These results are very promising and encourage for further research in this area.

Future work should focus on improving the approach for extracting features from data,

gathering a larger quantity of data, combining the tool with other approaches for enhanced results, and exploting codes from real projects. The work could also be extended to other programming languages or types of vulnerabilities. Our project has been made available as a public Github repository alongside with datasets, trained models, and examples [77].

## 6.1 Future work

There are several unsolved issues for future research in this field of study. Firstly, the approach itself could probably be adjusted and improved, for instance, by optimizing the feature extraction process or more data collection as discussed earlier.

The 'Var' attribute values could be refined in order to match exactly the same pattern with the truly vulnerable codes in order to improve false positives cases from the model.

Our data set is very biased due to the fact that each source code line is a sample and there are more non-vulnerable lines than vulnerable ones. The data could be compiled and filtered more thoroughly, leading to much more representative samples. By collecting more data from different source, it could be ensured that a larger percentage of the samples actually contain more relevant examples.

Our approach could be experimented with different programming languages like PHP, C++, or C#. First, we need to see if the process of extracting features (data transformation) works well for those languages. If it does, we can train the model on datasets from those languages and compare it to other research. Additionally, we could use our approach on various type of applications, including web and mobile apps or software in specific fields with their unique security risks.

The approach could also be extended to provide alerts for bugs that may be present in the source code. There is a whole research area dealing with this problem, and the application of machine learning can surely offer valuable insights. The possibilities are vast, and a lot of work is still to be done in this area.

Finally, a subsequent expansion could involve building a fully functional vulnerability detection system with high usability that takes code as input and detect vulnerabilities. While a simple prototype tool has already been constructed in this work, it has not been optimized for usability in daily programming. One option would be to create a plugin for an IDE or a highly customized command line tool.

# References


[1] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda, *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*, 1st. Addison-Wesley Professional, 2013, ISBN: 032193315X.

[2] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg, "Static code analysis to detect software security vulnerabilities - does experience matter?", in *2009 International Conference on Availability, Reliability and Security*, 2009, pages 804–810. DOI: 10.1109/ARES.2009.163.

[3] Alfred V. Aho, *Compilers: Principles, Techniques, and Tools*. Greg Tobin, 2007.

[4] Anastasios Stasinopoulos, Christoforos Ntantogian, and Christos Xenakis, "Commix: Detecting and exploiting command injection flaws", Nov. 2015.

[5] Anna L. Buczak and Erhan Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection", *IEEE*, vol. 18, no. 2, pages 1153–1176, 2016. DOI: 10.1109/COMST.2015.2494502.

[6] Greg Van Houdt, Carlos Mosquera, and Gonzalo Nápoles, "A review on the long short-term memory model", *Artificial Intelligence Review*, vol. 53, Dec. 2020. DOI: 10.1007/s10462-020-09838-1.

[7] Maad Mijwil, Adam Esen, and Aysar Alsaadi, "Overview of neural networks", vol. 1, page 2, Apr. 2019.

[8] 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*, 2012, pages 152–156. DOI: 10.1109/MINES.2012.202.

[9] Sebastiano Biondo, Emilio Ramos, Manuel Deiros, Juan Ragué, Javier Oca, Pablo Moreno, Leandre Farran, and Eduardo Jaurrieta, "Prognostic factors for mortality in left colonic peritonitis: A new scoring system", *Journal of the American College of Surgeons*, vol. 191, pages 635–42, Jan. 2001. DOI: 10.1016/S1072-7515(00)00758-4.

[10] Boyd Carl; Tolson Mary Ann; Copes Wayne S., "Evaluating trauma care: The triss method", *The Journal of Trauma: Injury, Infection, and Critical Care*, pages 370–378, Apr. 1987.

[11] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer, "Smote: Synthetic minority over-sampling technique", *Journal of artificial intelligence research*, vol. 16, pages 321–357, 2002.

[12] Checkmarx. "Checkmarx". (), [Online]. Available: https://www.checkmarx.com/.

[13] Christopher M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1st ed. Springer, 2007, ISBN: 0387310738. [Online]. Available: http://www.amazon.com/Pattern-Recognition-Learning-Information-Statistics/dp/0387310738%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0387310738.

[14] Common Vulnerabilities and Exposures. "Cve-2021-42264: Adobe premiere pro 15.4.1". (), [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42264.

[15] CVE. "Common vulnerabilities and exposures". (Aug. 2023), [Online]. Available: https://cve.mitre.org/.

[16] Common Weakness Enumeration. "Cwe-476: Null pointer dereference". (Jul. 2006), [Online]. Available: https://cwe.mitre.org/data/definitions/476.html.

[17] Common Weakness Enumeration. "2022 cwe top 25 vulnerabilities ranking". (Feb. 2023), [Online]. Available: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[18] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg, "Static code analysis to detect software security vulnerabilities - does experience matter?", in *2009 International Conference on Availability, Reliability and Security*, 2009, pages 804–810. DOI: 10.1109/ARES.2009.163.

[19] Douglas Thain, *Introduction to Compilers and Language Design*. Paperback ISBN: 979-8-655-18026-0, 2020.

[20] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997, Available at www.dspguide.com. [Online]. Available: http://www.dspguide.com.

[21] European Union Agency for Cybersecurity. "Risk-management-inventory". (Jan. 2023), [Online]. Available: https://www.enisa.europa.eu/topics/risk-management/current-risk/risk-management-inventory/glossary#G52.

[22] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck, "Generalized vulnerability extrapolation using abstract syntax trees", Dec. 2012, pages 359–368. DOI: 10.1145/2420950.2421003.

[23] Flawfinder. "Flawfinder". (), [Online]. Available: http://www.dwheeler.com/flawfinder.

[24] Chris Larsen. "Opentsdb". (), [Online]. Available: https://github.com/OpenTSDB/opentsdb.

[25] Christophe Tafani-Dereeper. "Vulnerable java application". (), [Online]. Available: https://github.com/DataDog/vulnerable-java-application.

[26] David M. "Dbus-java". (), [Online]. Available: https://github.com/hypfvieh/dbus-java.

[27] Sri Harsha. "Selenium". (), [Online]. Available: https://github.com/SeleniumHQ/selenium.

[28] Sridhar Gopinath. "Null-pointer-dereference". (), [Online]. Available: https://github.com/sridhargopinath/null-pointer-dereference.

[29] Alibaba Syclover. "Java sec code". (), [Online]. Available: https://github.com/JoyChou93/java-sec-code/tree/master.

[30] Utku Onur Şahin. "Os command injection". (), [Online]. Available: https://github.com/utkuonursahin/injections.

[31] Vishal Nehra. "Amaze file manager". (), [Online]. Available: https://github.com/TeamAmaze/AmazeFileManager.

[32] Will Hack. "Os command injection vulnerability in java spring". (), [Online]. Available: https://gist.github.com/pich4ya/c046378ed8ae8705147dad7ec9e0f6fc.

[33] David Molnar Patrice Godefroid Michael Y. Levin, "Whitebox fuzzing for security testing", *IEEE*, vol. 55, pages 40–44, Mar. 2012. DOI: https://doi.org/10.1145/2093548.2093564.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[35] Harry Zhang, "The optimality of naive bayes", in *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*, May 17-19, 2004 (Miami Beach, Florida, USA), Valerie Barr and Zdravko Markov, Eds., AAAI Press, 2004.

[36] IBM. "What is logistic regression?" (Dec. 2022), [Online]. Available: https://www.ibm.com/topics/logistic-regression.

[37] Ian H. Witten and Eibe Frank, *Data Mining: Practical Machine Learning Tools and Techniques* (Morgan Kaufmann Series in Data Management Systems), 2nd. Morgan Kaufmann, 2005.

[38] scikit learn. "Imbalanced-learn". (), [Online]. Available: https://github.com/scikit-learn-contrib/imbalanced-learn.

[39] ImmuniWeb, "Null pointer dereference [cwe-476]", 2012. [Online]. Available: https://www.immuniweb.com/vulnerability/null-pointer-dereference.html.

[40] Amer Zayegh and Nizar Albassam, "Neural network principles and applications", in Nov. 2018, ISBN: 978-1-78984-540-2. DOI: 10.5772/intechopen.80416.

[41] Daniel Berrar, "Cross-validation", in Jan. 2018, ISBN: 9780128096338. DOI: 10.1016/B978-0-12-809633-8.20349-X.

[42] Shashank Singh and Amrita Chaturvedi, "Applying deep learning for discovery and analysis of software vulnerabilities: A brief survey", in Jun. 2020, pages 649–658, ISBN: 978-981-15-4031-8. DOI: 10.1007/978-981-15-4032-5_59.

[43] ISO. "Iso/iec 27005:2008 information technology — security techniques — information security risk management". (Feb. 2023), [Online]. Available: https://www.iso.org/standard/42107.html.

[44] Marshall JC; Cook DJ; Christou NV; Bernard GR; Sprung CL; Sibbald WJ, "Multiple organ dysfunction score: A reliable descriptor of a complex clinical outcome", *Critical Care Medicine*, vol. 23, pages 1638–1652, Oct. 1995. DOI: 10.1097/00003246-199510000-00007.

[45] Jean-Roger Le Gall, Stanley Lemeshow, and Fabienne Saulnier, "A New Simplified Acute Physiology Score (SAPS II) Based on a European/North American Multicenter Study", *JAMA*, vol. 270, no. 24, pages 2957–2963, Dec. 1993, ISSN: 0098-7484. DOI: 10.1001/jama.1993.03510240069035. eprint: https://jamanetwork.com/journals/jama/articlepdf/409979/jama_270_24_035.pdf. [Online]. Available: https://doi.org/10.1001/jama.1993.03510240069035.

[46] Jerome H Friedman and Charles B Roosen, "An introduction to multivariate adaptive regression splines", *Statistical Methods in Medical Research*, vol. 4, no. 3, pages 197–217, 1995, PMID: 8548103. DOI: 10.1177/096228029500400303. eprint: https://doi.org/10.1177/096228029500400303. [Online]. Available: https://doi.org/10.1177/096228029500400303.

[47] J. Ross Quinlan, "Cc4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993", *Machine Learning*, vol. 16, 235–240, Sep. 1993. DOI: 10.1007/BF00993309. [Online]. Available: https://doi.org/10.1007/BF00993309.

[48] J. Bruin. "The javascript code quality tool". (Dec. 2022), [Online]. Available: http://JSLint.com/.

[49] J. Viega, J.T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code", in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000, pages 257–267. DOI: 10.1109/ACSAC.2000.898880.

[50] Jeremy Kolter and Marcus Maloof, "Learning to detect and classify malicious executables in the wild.", *Journal of Machine Learning Research*, vol. 6, pages 2721–2744, Dec. 2006.

[51] Jorrit Kronjee, Arjen Hommersom, and Harald Vranken, "Discovering software vulnerabilities using data-flow analysis and machine learning", in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES '18, Hamburg, Germany: Association for Computing Machinery, 2018, ISBN: 9781450364485. DOI: 10.1145/3230833.3230856. [Online]. Available: https://doi.org/10.1145/3230833.3230856.

[52] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon, "Mining fix patterns for findbugs violations", *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pages 165–188, 2021. DOI: 10.1109/TSE.2018.2884955.

[53] J.R. Larus, T. Ball, Manuvir Das, R. DeLine, M. Fahndrich, J. Pincus, S.K. Rajamani, and R. Venkatapathy, "Righting software", *IEEE Software*, vol. 21, no. 3, pages 92–100, 2004. DOI: 10.1109/MS.2004.1293079.

[54] L. Breiman, "Random forests", *CA 94720*, 2001.

[55] anikaseth98. "Role of log odds in logistic regression". (Feb. 2023), [Online]. Available: https://www.geeksforgeeks.org/role-of-log-odds-in-logistic-regression/.

[56] Wikipedia. "Maximum a posteriori estimation". (Feb. 2023), [Online]. Available: https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation.

[57] Michael Howard, David Leblanc, and Brian Valentine, *Writing Secure Code*. USA: Microsoft Press, 2001, ISBN: 0735615888.

[58] Mohammed Zagane, Mustapha Abdi, and Mamdouh Alenezi, "Deep learning for software vulnerabilities detection using code metrics", *IEEE Access*, vol. PP, pages 1–1, Apr. 2020. DOI: 10.1109/ACCESS.2020.2988557.

[59] Mukesh Kumar Gupta, M.C. Govil, and Girdhari Singh, "Static analysis approaches to detect sql injection and cross site scripting vulnerabilities in web applications: A survey", in *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, 2014, pages 1–5. DOI: 10.1109/ICRAIE.2014.6909173.

[60] Nadeem Shah and Mohammed Farik, "Ransomware-threats, vulnerabilities and recommendations", *International Journal of Scientific & Technology Research*, vol. 6, pages 307–309, Jun. 2017.

[61] Alexandru Niculescu-Mizil and Rich Caruana, "Predicting good probabilities with supervised learning", Jan. 2005, pages 625–632. DOI: 10.1145/1102351.1102430.

[62] M Kologlu 1; D Elker; H Altun; I Sayek. "Validation of mpi and pia ii in two different groups of patients with secondary peritonitis". (Feb. 2023), [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/11268952/.

[63] Wikipedia. "Natural language processing". (Feb. 2023), [Online]. Available: https://en.wikipedia.org/wiki/Natural_language_processing.

[64] NVD. "National vulnerability database - statistics search". (Feb. 2023), [Online]. Available: https://web.nvd.nist.gov/view/vuln/statistics.

[65] Geoffrey I. Webb. "Overfitting". (Jan. 2023), [Online]. Available: https://link.springer.com/referenceworkentry/10.1007/978-0-387-30164-8_623.

[66] Martin Hamant. "Ow2: Spoon control-flow". (Feb. 2023), [Online]. Available: https://gitlab.ow2.org/spoon/spoon/-/tree/master/spoon-control-flow.

[67] OWASP. "The ten most critical web application security vulnerabilities". (Feb. 2023), [Online]. Available: http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf.

[68] OWASP. "Owasp top ten". (Feb. 2023), [Online]. Available: https://owasp.org/www-project-top-ten/.

[69] Pietro Oliva. "Cve-2020-10231 tplink webcam null-pointer-dereference vulnerability". (Feb. 2023), [Online]. Available: https://cxsecurity.com/issue/WLB-2020040002.

[70] DeKok. "A limited problem scanner for c source files". (Jan. 2023), [Online]. Available: http://deployingradius.com/pscan/.

[71] rahulkhinchi7, "Abstract syntax tree", *geeksforgeeks*, 2021.

[72] Checkmarx. "Rough-auditing-tool-for-security". (), [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[73] Yuzhu Ren, Jiangtao Zhao, and Cao Zhang, "The propagation strategy model of taint analysis", *Journal of Physics: Conference Series*, vol. 1486, no. 4, page 042 040, 2020. DOI: 10.1088/1742-6596/1486/4/042040. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1486/4/042040.

[74] REN Yuzhu; ZHANG Youwei; AI Chengwei, "Survey on taint analysis technology", *Journal of Computer Applications*, vol. 39, no. 8, pages 2302–2309, 2019.

[75] scikit learn. "Cross-validation: Evaluating estimator performance". (), [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html.

[76] SAMATE. "Software assurance metrics and tool evaluation project main page". (Feb. 2023), [Online]. Available: https://samate.nist.gov/Main_Page.html.

[77] sanaconte, *Softwareweaknesses detection using static-code analysis and machine learning techniques*, https://github.com/sanaconte/SoftwareWeaknessesDetection, 2023.

[78] scikit learn. "1.4. support vector machines". (Feb. 2023), [Online]. Available: https://scikit-learn.org/stable/modules/svm.html.

[79] scikit learn. "Working with text data". (Jul. 2023), [Online]. Available: https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html.

[80] Seyed Ghaffarian and Hamid Reza Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey", *ACM Computing Surveys*, vol. 50, pages 1–36, Aug. 2017. DOI: 10.1145/3092566.

[81] Sebastian Anthony. "Shellshock: A deadly new vulnerability that could lay waste to the internet". (Feb. 2023), [Online]. Available: https://www.extremetech.com/computing/190959-shellshock-a-deadly-new-vulnerability-that-could-lay-waste-to-the-internet.

[82] Jason Brownlee. "Smote for imbalanced classification with python". (Jan. 2021), [Online]. Available: https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

[83] Sofia Bekrar; Chaouki Bekrar; Roland Groz; Laurent Mounier, "A taint based approach for smart fuzzing", *IEEE*, vol. 1, no. 1, 818–825, 2012.

[84] Zoltán Somogyi, "Performance evaluation of machine learning models", in *The Application of Artificial Intelligence: Step-by-Step Guide from Beginner to Expert*. Cham: Springer International Publishing, 2021, pages 87–112, ISBN: 978-3-030-60032-7. DOI: 10.1007/978-3-030-60032-7_3. [Online]. Available: https://doi.org/10.1007/978-3-030-60032-7_3.

[85] Jason Brownlee. "How to encode text data for machine learning with scikit-learn". (Jul. 2023), [Online]. Available: https://machinelearningmastery.com/prepare-text-data-machine-learning-scikit-learn/.

[86] Tom M Mitchell, *Machine learning*. McGraw-hill New York, 1997, vol. 1.

[87] Vijay Ganesh, Tim Leek, and Martin Rinard, "Taint-based directed whitebox fuzzing", Jan. 2009, pages 474–484. DOI: 10.1109/ICSE.2009.5070546.

[88] Vladimir Vapnik, *The Nature of Statistical Learning Theory*. Springer: New York, 2000.

[89] Laura Wartschinski, *Detecting software vulnerabilities with deep learning*, This is a master's thesis of Laura Wartschinski from Humboldt University of Berlin, Dec. 2019.

[90] Weilin Zhong. "Command injection". (Feb. 2023), [Online]. Available: https://owasp.org/www-community/attacks/Command_Injection.

[91] Wenhui Jin, Sami Ullah, Dongmin Yoo, and Heekuck Oh, "Npdhunter: Efficient null pointer dereference vulnerability detection in binary", *IEEE Access*, vol. 9, pages 90 153–90 169, 2021. DOI: 10.1109/ACCESS.2021.3091209.

[92] Wikipedia. "European union agency for cybersecurity". (Jan. 2023), [Online]. Available: https://en.wikipedia.org/wiki/European_Union_Agency_for_Cybersecurity.

[93] Wikipedia. "Word2vec". (Feb. 2023), [Online]. Available: https://en.wikipedia.org/wiki/Word2vec.

[94] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman, "The matter of heartbleed", in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, 475–488, ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. [Online]. Available: https://doi.org/10.1145/2663716.2663755.

[95] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection", Jan. 2018. DOI: 10.14722/ndss.2018.23165.