UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Processing Web Applications using NLP for Vulnerability Detection and Explanation

Jorge Manuel Gomes Guerreiro

Mestrado em Informática

Dissertação orientada por:
Ibéria Vitória de Sousa Medeiros

2025

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Profª. Ibéria Medeiros, who was more than a supervisor. She was a fundamental part of this project and helped me to overcome every challenge.

Without the presence of some wonderful people in my life, this work would not have unfolded as it did. I am eternally grateful to my mother, Maria, for the determination and strength she has always instilled in me. To my father, Paulo (*in memoriam*), for always believing in me. To my sister, Sofia, for her support and for all the good moments. To my dog, Zara, for always being there for me. To my grandparents, Maria and José, who have always done everything for me. I would also like to thank, in a special way, my girlfriend, Inês, whose support and encouragement were fundamental in the past years. I am also thankful to friends who made this journey easier, namely Vasco, Tiago, Mariana, and André. To all the other people not mentioned by name, which know they are important, I also wish to express my sincere gratitude.

This work was supported by P2030 through project I2DT, ref. COMPETE2030-FEDER-00389100, an ITEA4 European project (ref. 22025), and by FCT through the LASIGE Research Unit, ref. UID/00408/2023.

Dedicado aos meus pais.

Resumo

No mundo em que vivemos hoje, inúmeros serviços, das mais variadas áreas da nossa sociedade, são disponibilizados através de aplicações web, devido a diversos fatores inerentes ao mundo tecnológico em que estamos inseridos e que suportam a nossa atual forma de estar. A praticidade de utilização das mesmas através de um simples *smartphone* ou de um *laptop* é um desses fatores, que as torna um elemento essencial e valioso da vida moderna, devido aos dados dos utilizadores que são guardados em bases de dados e acedidos pelas mesmas. Na mesma via, as aplicações web tornam-se um alvo constante de ataques informáticos, que são possíveis devido à existência de vulnerabilidades no código fonte das mesmas. De acordo com a Open Web Application Security Project (OWASP), dentro das vulnerabilidades mais críticas e impactantes, encontram-se SQL Injection (SQLi) e Cross-Site Scripting (XSS). Aquando da ausência de sanitização e/ou validação de *inputs* fornecidos por utilizadores, os atacantes conseguem forjar *inputs* que permitem o acesso indevido a bases de dados ou mesmo executar código malicioso, que compromete a confidencialidade, integridade e disponibilidade dos dados. Para não permitir estes ataques, os desenvolvedores de aplicações web têm de garantir que os *inputs* são validados através de funções de sanitização, o que nem sempre acontece, devido a fatores como prazos limitados de desenvolvimento, falta de experiência, ou mesmo negligência.

Este problema de deteção de vulnerabilidades em aplicações web é cada vez mais relevante, sendo crítico nos dias de hoje. Estima-se que cerca de 75% das aplicações web estão programadas em PHP. E, apesar de já existirem vários avanços na deteção de vulnerabilidades, tanto com ferramentas de análise estática e com abordagens com *machine learning*, estes avanços continuam a apresentar diversas limitações. A complexidade crescente dos ataques torna insuficientes os métodos que se baseiam em regras fixas e, mesmo que, as ferramentas consigam identificar potenciais vulnerabilidades, a existência destas não é explicada de forma clara, o que dificulta a tarefa do programador em corrigi-las. Neste sentido, é necessário explorar metodologias que combinem deteção automática de vulnerabilidades com mecanismos de explicação das mesmas.

Neste contexto, esta dissertação propõe uma abordagem que visa utilizar técnicas de Processamento de Linguagem Natural (NLP) para detetar e explicar vulnerabilidades em código existentes no código de aplicações web desenvolvidas em PHP. Para isto, uma Linguagem Intermédia (IL) foi definida para permitir a tradução do código PHP para uma linguagem mais simples e generalista, por forma a que esta possa ser processada, eficientemente, por vários modelos de NLP. Esta IL tem o objetivo de ser uma linguagem mais genérica, que mantenha a lógica e a semântica do código

e assegure o fluxo dos dados original, de forma a que os padrões originais se mantivessem. Por exemplo, os nomes das variáveis, classes e funções existentes no código são normalizados para nomes genéricos, mas assegurado o fluxo dos dados entre estes, bem como a lógica e semântica existente no código.

Para detetar e explicar vulnerabilidades em código PHP decidimos utilizar duas famílias de modelos de NLP: modelos neuronais e modelos sequenciais. Os modelos neuronais, como o LSTM e Transformers, são os responsáveis pela classificação de cada fragmento de código em duas classes (vulnerável ou não vulnerável), devido à capacidade destes modelos de aprenderem padrões complexos. Os modelos sequenciais supervisionados, como o HMM e MEMM, são utilizados para aprenderem padrões em código anotado *token* a *token* (i.e., palavra a palavra) na nossa IL, para que estes posteriormente sejam capazes de anotar os tokens dos fragmentos de código que analisam e processam. Desta forma, estes modelos são capazes de explicar a propagação de dados, que podem ser ou não sanitizados. Para além do modelo sequencial supervisionado HMM, ainda utilizamos o HMM não-supervisionado ao nível de cada *token* como modelo classificativo, para avaliar a capacidade classificativa de um modelo sequencial.

A integração destes modelos deu origem à ferramenta VulNLan, cuja arquitetura foi desenhada em duas fases: (i) treino e teste, onde são treinados e *fine-tuned* todos os modelos com base em conjuntos de dados em IL, tanto anotados *token* a *token* (i.e., supervisionado), como não anotados (i.e., não supervisionado); (ii) deteção de vulnerabilidades, onde o novo fragmento de código é processado e traduzido de PHP para IL, classificado e explicado em detalhe. Para agregar os resultados dos diferentes classificadores, foi definida uma heurística baseada na eficácia (*accuracy*) relativa de cada modelo, atribuindo maior peso às previsões mais fiáveis, e uma tabela de decisão onde a classe escolhida pela maioria dos modelos é a passada para os modelos explicativos.

Para os diferentes modelos foram construídos conjuntos de dados (*datasets*) de treino a partir de um mesmo *dataset* fonte. Este *dataset* fonte proveu do Software Assurance Reference Dataset (SARD), do National Institute of Standards and Technology (NIST), que contém ficheiros PHP anotados a nível do ficheiro como *safe* ou *unsafe*, referentes à vulnerabilidade SQLi. Deste, extraímos um *dataset* balanceado, o qual traduzimos e utilizamos para construir dois tipos de *datasets* em IL: um *dataset* que contém os elementos normalizados com identificadores que os distinguem e um *dataset* sem identificadores onde todas as variáveis aparecem iguais, por exemplo. Para estudar a viabilidade destes *datasets*, decidimos estudar a similaridade dos mesmos, utilizando uma técnica de NLP chamada TF-IDF.

Os modelos de Transformers foram treinados e testados com ambos *datasets* IL (i.e., com e sem identificadores). No total, treinamos e testamos oito modelos de Transformers, devido ao facto de utilizarmos quatro modelos pré-treinados para cada um dos datasets. Para os modelos LSTM e HMM não-supervisionado, utilizamos o *dataset* que contém os identificadores. No caso do LSTM, treinámos três modelos diferentes, correspondendo a três otimizadores, enquanto no HMM não-supervisionado, treinámos um modelo para cada classe do conjunto (vulnerável e não vulnerável). Para os modelos sequenciais supervisionados, as instâncias do *dataset* IL sem identi-

ficadores foram anotadas *token* a *token* com estados definidos e que assumimos como adequados para demonstrar a propagação de *inputs* maliciosos ao longo do código. Todos os modelos foram treinados, testados e afinados (*fine-tuned*) diversas vezes e em ciclo, até atingirem resultados consistentes.

Os resultados dos modelos são promissores. Todos os modelos de Transformers apresentaram uma *accuracy* acima dos 92% e precisões superiores a 96%, o que mostra uma tendência de mais falsos negativos do que falsos positivos, que se manteve ao longo de todos os modelos. No entanto, apesar desta tendência ténue, estes valores denotam confiança na classificação efetuada por estes quatro modelos. Para os LSTM, atingimos resultados certeiros em cerca de 95% dos casos nos melhores modelos, tendo em conta que um dos três otimizadores não convergiu. O HMM não-supervisionado obteve uma prestação mais baixa do que os outros modelos classificativos, atingindo resultados na ordem dos 70% para todas as métricas. Os modelos explicativos apresentaram resultados semelhantes na explicação de instâncias vulneráveis (com *accuracy* de 90% e F1-score de 92%), onde na classe não vulnerável, o HMM supervisionado foi superior (com *accuracy* de 90%) ao MEMM (com *accuracy* de 78%).

Na avaliação da ferramenta VulNLan como um todo, os modelos neuronais apresentaram precisões a rondar os 96%, o que confirma a capacidade destes modelos de se adaptarem ao código em IL. A heurística apresentou resultados associados aos melhores modelos classificativos, como esperado. Os modelos explicativos demonstraram sucesso na identificação da propagação de *inputs* maliciosos, assim como na sanitização dos mesmos, cumprindo o objetivo de formar um *output* interpretável. No global, a ferramenta VulNLan obteve uma eficácia global de 96%, com um *F1-score* de 96%.

Concluindo, a nossa metodologia, explicada em detalhe nesta dissertação, demonstra que a utilização de modelos NLP para processamento de uma IL, mesmo que generalista, é capaz de detetar e explicar vulnerabilidades em código fonte de aplicações web. Melhorar a deteção e explicação de vulnerabilidades, permitindo que os programadores desenvolvam e corrijam código de forma a proteger os utilizadores, é o principal objetivo da investigação que poderá dar continuidade a este trabalho.

Palavras-chave: Vulnerabilidades, Aplicações Web, Modelos de NLP, Explicabilidade, Segurança de Software.

Abstract

Web applications increasingly store and process valuable user data, which makes flaws in their source code a direct threat to users' privacy and security. Attackers commonly exploit vulnerabilities such as SQL Injection (SQLi) and Cross-Site Scripting (XSS), often taking advantage of missing sanitization or validation of user inputs. Existing static analysis and machine-learning tools can flag potential faults, but rule-based methods struggle with evolving attack complexity and, crucially, provide explanations that are insufficient for developers to remediate the issues. This dissertation proposes a novel Natural Language Processing (NLP)-based methodology for detecting and explaining vulnerabilities in PHP web applications. We introduce an Intermediate Language (IL) produced by a pre-processing and translation tool, PHP2IL, which normalises PHP source code while preserving program logic and data flow. Pre-processing removes biasing elements (comments, literal strings) and obfuscates identifiers, enabling NLP models to learn structural and semantic patterns rather than surface artefacts. Two complementary families of models are employed: neural models (LSTM, Transformers) perform code snippet classification (vulnerable or not vulnerable), while sequential supervised models (HMM, MEMM) are trained token-by-token on annotated IL to trace tainted data propagation and explain where sanitization is missing. However, before the explanation, an heuristic aggregates the classification models outputs, weighting predictions by model accuracy, and forwards the code in analysis to the corresponding explanatory models of the determined final class. The approach was implemented in the VulNLan tool, and we evaluated it with the NIST SARD dataset focused on SQLi in PHP. Neural classifiers achieved approximately 96% precision and when combined in the *VulNLan* tool the ensemble reached an accuracy of 0.960 with an F1-score of 0.962. The sequential explanation models successfully identified taint propagation and sanitization points, producing interpretable outputs. These results show that applying NLP to an IL representation can both detect and explain vulnerabilities in PHP source, closing the gap between automated detection and actionable remediation – helping developers fix code and protecting end users.

Keywords: Vulnerabilities, Web applications, NLP models, Explainability, Software security.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AF Activation Function.

AI Artificial Intelligence.

ALPAC Automatic Language Processing Advisory Committee.

AST Abstract Syntax Tree.

CPU Central Processing Unit.

CRF Conditional Random Field.

CSV Comma-Separated Values.

CWE Common Weakness Enumeration.

GNN Graph Neural Network.

GPT Generative Pre-trained Transformer.

GPU Graphics Processing Unit.

HMM Hidden Markov Model.

HTML HyperText Markup Language.

IDS Intrusion Detection System.

IDs Identifiers.

IL Intermediate Language.

LSTM Long Short-Term Memory.

MEMM Maximum-Entropy Markov Model.

ML Machine Learning.

MT Machine Translation.

NIST National Institute of Standards and Technology.

NLP Natural Language Processing.

NN Neural Network.

OWASP Open Web Application Security Project.

PHP Hypertext Preprocessor.

PoS Part-of-Speech.

RAM Random Access Memory.

RNN Recurrent Neural Network.

SARD Software Assurance Reference Dataset.

SQL Structured Query Language.

SQLi SQL Injection.

SS Sensitive Sinks.

TF-IDF Term Frequency–Inverse Document Frequency.

XSS Cross-Site Scripting.

Chapter 1

Introduction

This document presents a Master's dissertation submitted to the Faculty of Sciences at the University of Lisbon. The core objective of this research is to advance the state-of-the-art in software security by leveraging Natural Language Processing (NLP) techniques to accurately identify and explain vulnerabilities within the source code of web applications, specifically focusing on those written in PHP.

1.1 Motivation

Information security, defined by the National Institute of Standards and Technology (NIST) as the protection of information systems from unauthorised actions in order to ensure confidentiality, integrity, and availability [43], is fundamental to modern privacy and infrastructure [6].

Despite heightened awareness, the cyber threat landscape continues to escalate. Since 2020, the annual number of data breach victims has ranged from 300 million to 425 million, with reported cyberattacks increasing significantly from 877 in 2020 to 2,365 in 2023 [19]. This trend reveals growing complexity in cybercrime, which in turn imposes considerable financial burdens on organisations. The global average total cost of a data breach has risen to $4.88 million in 2024 [18], necessitating the exploration of more sophisticated defence mechanisms. Critically, organisations that extensively leverage Artificial Intelligence (AI) have reported a 45.6% reduction in their data breach costs [18]. Given that 73% of large organisations (20,000+ employees) have now implemented Generative AI tools [20], and despite a knowledge gap among professionals [19], this fast integration confirms that AI technology is rapidly shaping the future of cybersecurity defence.

A major focus of cyberattacks remains web applications. Applications developed using PHP are particularly targeted as the language serves as the server-side technology for approximately 75% of web applications currently available [57]. This widespread adoption makes rigorous security testing essential, as any vulnerability carries significant risk. Among the various vulnerability classes, injection vulnerabilities remain the most exploited, with SQL Injection (SQLi) being one of the most impactful, as highlighted by OWASP [46].

In recent years, efforts to detect PHP vulnerabilities have led to the creation of static analysis tools integrating Machine Learning (ML), such as DEKANT [39], MERLIN [11], and WAP [38].

Although these tools have demonstrated commendable results, the relentless evolution of technology, the growing complexity of threats, and the aggressive adoption of advanced AI techniques across the industry collectively demand the development of a new generation of vulnerability detection tools. The rapid pace of innovation in AI provides an opportunity to create models capable of automating tasks previously deemed difficult, such as sophisticated vulnerability detection in source code.

1.2 Objectives

To tackle the problem identified, this dissertation has the goal of contributing to the detection and explanation of vulnerabilities in web applications, proposing an approach with no rule dependence and entirely supported by NLP models. This dissertation presents a novel, Natural Language Processing (NLP)-based approach for the robust detection and subsequent explanation of PHP vulnerabilities. The approach is defined by the integration of diverse NLP models, specifically combining neural networks (e.g., Transformers, LSTMs) with sequential models (e.g., Hidden Markov Models).

The core mechanism involves processing PHP source code that has been translated into an Intermediate Language (IL). The neural networks perform the high-level classification, indicating the absence or presence of a vulnerability, while the sequential models are designed to interpret the results of these "black-box" classifiers, providing an explanation and pinpointing the exact location of the vulnerability within the code.

The use of an Intermediate Language serves to represent the code in a generic, semantically-preserving manner, making it suitable for training diverse NLP models. Crucially, the translation process includes meticulous pre-processing to clean the code of irrelevant data, such as comments, which could otherwise induce model overfitting based on information that does not reflect the code's true logic.

A preliminary evaluation of the neural networks used in this approach for SQL Injection (SQLi) vulnerability detection showed promising results, demonstrating that the models correctly interpret IL code and are able to detect vulnerabilities with a precision of 96%.

1.3 Contributions

This dissertation makes the following contributions to the field of software security:

1. Study of vulnerability detection models and tools, as well as the abstract representation of specific source code.

2. A novel, explainable NLP-based approach for vulnerability identification in PHP source code, combining deep learning and sequential models to provide both detection and token-level localisation.

3. The design and specification of an Intermediate Language (IL) for code representation, preserving semantic and logical meaning while facilitating NLP model training.
4. A comprehensive architectural proposal integrating general code snippet classification with token-level classification models, with the final classification supported by an accuracy-weight heuristic.
5. VulNLan, a prototype of the approach and its evaluation, demonstrating the models' capacity to correctly process and interpret IL for vulnerability detection.

The work conducted on this dissertation resulted in peer-reviewed publications and presentations, specifically:

- A conference paper, titled "Processing Web Applications using NLP for Vulnerability Identification", presented at the 20th European Dependable Computing Conference (EDCC), Lisbon, Portugal, 2025 [13];
- A poster with the same paper title, presented at EDCC 2025, which won a Distinguished Poster Award.
- A poster presentation outlining the work at the 10th LASIGE Workshop, 2025.

1.4 Structure of the Document

The remainder of this document is organised as follows:

- **Chapter 2 (Background and Related Work)** provides the theoretical foundation for this study, reviewing Web Application Vulnerabilities, the relevant principles of Natural Language Processing (NLP), including Neural Network Models and Sequence Models, and discusses existing Vulnerability Detection Tools.
- **Chapter 3 (Proposed Solution & Design)** details the research methodology, outlining the key Challenges and presenting the VulNLan tool architecture. This chapter covers the process of Translating PHP to Intermediate Language, the Training and Testing Phase (including the Dataset and Models training), and the Vulnerability Detection Phase, with a focus on the underlying Heuristic.
- **Chapter 4 (Implementation)** specifies the technical realisation of the solution, detailing the creation of the IL Translator, the Datasets construction, the Training and fine-tuning models (both for classification and explanation), the development of the VulNLan tool, and the process of Classification and Interpretation.
- **Chapter 5 (Evaluation & Results)** presents an assessment of the proposed solution, including a Datasets Analysis, the Experimental Setup, the observed Results for both classification and explanation models, a Comparative analysis, the VulNLan tool evaluation, and a final Discussion of the strengths and limitations.

- **Chapter 6 (Conclusion)** summarises the dissertation's key findings, reaffirms the contributions, and suggests avenues for Future work.

Chapter 2

Background and Related Work

This chapter begins with an overview of software vulnerabilities, showcasing two types of injection vulnerabilities. It then presents a historical summary of natural language processing, tracing its evolution from the mid-20th century to the present, with an emphasis on deep learning. Subsequently, the chapter explores sequence models and the Viterbi decoding algorithm, which is used with the models. Finally, it reviews relevant projects and state-of-the-art tools that are advancing the field of vulnerability detection through Natural Language Processing (NLP) techniques and ML approaches based on intermediate code representations.

2.1 Web Application Vulnerabilities

Vulnerabilities in web applications are based on user inputs. When a web application is built incorrectly or is not maintained and updated, there is a possibility that forged user inputs are able to compromise the web application, by stealing data, deleting databases, and other malicious activities. The vulnerability is present in the lack of sanitization before the input reaches Sensitive Sinks (SS). SS are the functions that communicate with the critical systems, such as databases, file systems, and others.

SQLi remains one of the most critical and commonly exploited vulnerabilities in web applications, as highlighted by Open Web Application Security Project (OWASP) [46]. This vulnerability arises when user inputs are improperly handled and passed directly into an Structured Query Language (SQL) query without appropriate sanitization or validation. The unsanitized input can be used to inject malicious SQL commands and metacharacters into the query, which are then executed by the database. These malicious inputs could allow attackers to bypass authentication, retrieve sensitive data, modify database contents, or even delete data, leading to severe consequences for both the application and its users. In Hypertext Preprocessor (PHP), SQLi often occurs when dynamic queries are constructed by concatenating user input directly into the query string, but can also happen if the developer uses an older sanitization function that is ineffective against all types of inputs.

In lines 2 and 3 of Listing 2.1, both variables `$username` and `$password` receive inputs from the user. These inputs are concatenated into a query string (in line 4) that is executed in the

database in line 5. The inputs are missing effective sanitization, i.e., the process of ensuring that the input does not contain any harmful characters. With this lack of sanitization, unauthorized commands could be executed in the database.

```
1 <?php
2 $username = $_GET['username'];
3 $password = $_GET['password'];
4 $query = "SELECT * FROM users WHERE username = '$username'
      AND password = '$password'";
5 $result = $db->query($query);
6 ?>
```

Listing 2.1: Example of SQL Injection Vulnerability.

Cross-Site Scripting (XSS) vulnerabilities are another common security concern in web development, and they can lead to a wide range of attacks on end-users. It is different from SQLi because it allows attackers to inject malicious scripts into web pages viewed by other users, which are executed in the context of the victim's browser instead of attacking the database. XSS vulnerabilities can generally be classified into three major categories: Stored XSS, Reflected XSS and DOM-based XSS; however, we only focus on the first two categories.

Stored XSS occurs when the malicious input provided by the attacker is stored on the server or database and then rendered by the web application whenever this data is later accessed by a user. A common scenario involves user comments or profile fields, where malicious script tags are entered, and these are subsequently stored and displayed on a page without proper sanitization. To prevent Stored XSS, developers should ensure that all user input is validated, sanitized, and escaped before being rendered on the page, without the possibility of malicious input injection in these input fields.

```
1  <?php
2  if ($_SERVER['REQUEST_METHOD'] === 'POST') {
3      $comment = $_POST['comment'];
4      $db->query("INSERT INTO comments (content) VALUES ('$comment')");
5  }
6  $result = $db->query("SELECT content FROM comments");
7  while ($row = $result->fetch_assoc()) {
8      echo "<p>" . $row['content'] . "</p>";
9  }
10 ?>
```

Listing 2.2: Example of Stored XSS Vulnerability.

In the vulnerable code presented in Listing 2.2, the user input in line 3 is stored in the database, in this example, as a comment to be displayed on the page. This comment accepts every type of string, benign or malign, not going through any kind of validation or sanitization function. The webpage loaded after this field of the database being compromised would then execute the code stored (lines 6 to 8).

Reflected XSS, on the other hand, does not involve persistent storage of the malicious payload. Instead, the attacker injects a script into a URL or a form field that is reflected back in the server response, often without proper validation. To mitigate Reflected XSS, input validation and output

encoding are critical. It is important to validate user input to ensure that it does not contain executable code and to escape special characters (such as <, >, &, and ") in the output before rendering it in the browser.

```
1 <?php
2 $search = $_GET['q'];
3 echo "<p>Results for: $search</p>";
4 ?>
```

Listing 2.3: Example of Reflected XSS Vulnerability.

In the vulnerable code presented in Listing 2.3, the variable `$search` receives the user input that can contain a script that is executed in line 3. An example of exploitation of Listing 2.3 is an input that starts with `</p>`, followed by malicious HyperText Markup Language (HTML) code, or a malicious script that compromises the web application. This vulnerability is also a consequence of the lack of user input validation and sanitization.

2.2 Natural Language Processing

NLP was introduced in the middle of the twentieth century as a research field focused on representing the meaning of words and sentences, enabling computers to retrieve information, translate text, and perform other tasks [24]. Initially, a research area known as Machine Translation (MT) was financed by the US Government, but after the Automatic Language Processing Advisory Committee (ALPAC) report in 1966, the financing of MT was reduced, and many researchers moved into NLP with the motivation that "something practical could be achieved" [3]. This transition brought innovation to NLP [24].

During this period of innovation, many approaches to processing text were developed. Until the 1980s, NLP systems were built using manually written linguistic rules, explicitly encoding grammatical knowledge, morphology, and syntax, and typically using custom-built parsers. One key limitation of such systems was their reliance on controlled vocabularies. As exemplified in [58], the state-of-the-art at the time was based on parse trees, which assisted in Part-of-Speech (PoS) identification, as show in Figure 2.1.

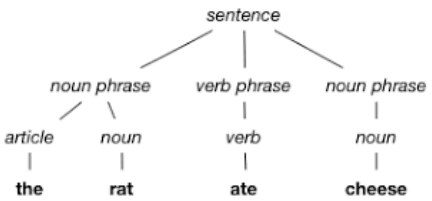

Figure 2.1: Parse tree for PoS tagging.

In some tasks, predominantly translation, systems would translate word by word through direct lexical correspondence, followed by morphological and syntactic rules mapping the sentence structure from one language to another [17].

One of the most influential early techniques in Natural Language Processing was introduced by Karen Spärck Jones in 1972 [23]. Known as Term Frequency–Inverse Document Frequency (TF-IDF), this method provided an innovative way to represent documents numerically based on the importance of their terms. The main idea is that the relevance of a term depends both on how often it appears in a given document (i.e., its *term frequency*) and on how rare it is across the entire collection (i.e., its *inverse document frequency*).

Formally, the term frequency of a token t in a document d is defined in 2.1:

$$\mathrm{tf}_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d) > 0, \\ 0 & \text{otherwise}, \end{cases} \tag{2.1}$$

where $\mathrm{count}(t,d)$ is the raw number of occurrences of term t in document d. The logarithm compresses large counts, mitigating the influence of very frequent terms.

The inverse document frequency, shown in 2.2, reflects the discriminative power of a term in the corpus. Given a collection of N documents and letting df_t denote the number of documents in which t occurs, IDF is expressed as:

$$\mathrm{idf}_t = \log_{10} \frac{N}{df_t}. \tag{2.2}$$

This value is higher for rare terms and lower (or zero) for terms that appear in nearly all documents.

Finally, the TF-IDF weight of term t in document d is obtained as the product of the two measures, as expressed in 2.3:

$$\text{tf-idf}(t,d) = \mathrm{tf}_{t,d} \cdot \mathrm{idf}_t. \tag{2.3}$$

The resulting TF-IDF representation has been foundational in Information Retrieval and later NLP applications, as it provides a vectorial encoding of documents that balances local and global term importance.

With the advancement of technology in general, Neural Network (NN)-based models emerged, such as Long Short-Term Memory (LSTM) [16]. These were also capable of learning patterns from data, further advancing the state of the art in NLP.

The most recent major breakthrough in this field came with the publication of the paper "Attention Is All You Need", which introduced attention mechanisms, which are the foundation of transformer architectures [56]. When combined with deep neural networks, transformers have outperformed all previous NLP models and have since demonstrated significant utility across a variety of tasks.

2.2.1 Neural Network Models

In the field of ML, NN emerged as computational models composed of nodes and the connections between them [50], as shown in Figure 2.2. Deep Learning refers to a subset of these networks that contain several hidden layers. Usually, they have input nodes and output nodes. In the middle, hidden layers that have parameters called weights and biases. Weights are multiplication parameters, while biases are addition ones [23, 51]. It starts with unknown parameter values that

are estimated when we fit the NN to a dataset using a method called *backpropagation*, that uses the chain rule to calculate the respective derivatives and inputs these derivatives into the Gradient Descent algorithm to optimize the parameters [51, 30]. These calculations also depend on an Activation Function (AF), that introduces non-linearity, and can vary with the chosen AF (e.g., ReLU, sigmoid, softmax, tanh, and more) [12]. These simple NNs have an issue within their arquitecture; they do not function with sequential data composed with input values of different size (e.g., natural language), because they have a fixed input size due to the lack of temporal memory [9].

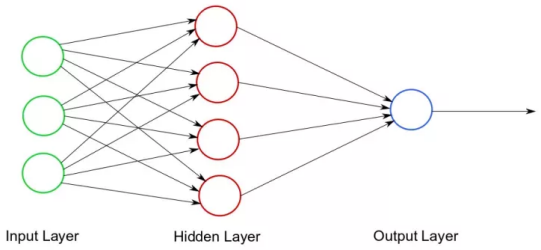

Figure 2.2: A simple neural network diagram.

To fix this problem, Recurrent Neural Network (RNN) were introduced, which added a feedback loop that is able to unfold the NN in order to handle sequential data [9]. But the RNN are also target of vanishing/explosion gradient problems. That happens when optimizing a parameter with the weights - calculating the lowest value for the Loss Function (which usually is an AF) - very small steps or very large steps are taken, respectively. For the vanishing problem, it hits the maximum of steps before finding the optimal value, and for the explosion problem this causes it to overshoot the minimum of the loss function, per example, 100 input values mean that the RNN unfolds 100 times, the step size would be weight raised to 100 power [5, 16].

LSTM is an NN that has two paths, one for Long-Term memories and one for Short-Term memories. With these paths, it is not the target of the RNN vanishing and explosion gradient problems. The LSTM selects: which information is important (input gate), by combining the short-term memory and the input to create a potential long-term memory and the percentage of the potential memory to remember, this determines how the long-term memory should be updated; which information should be discarded (forget gate), this determines what percentage of the Long-Term memory will be remembered; and which information it should share (output gate), in other words, updates the short-term memory by using new long-term memory as input to the tanh AF, which calculates the potential short-term memory. And it has to decide the percentage of potential short-term memory to remember using a sigmoid AF [16]. In Figure 2.3 these relations with the gates are represented, the forget gate works inside the memory cells, the input units can be any sequential data, with any size, and the output units are the final values of the network, that can be used for classification or prediction tasks.

Transformers is an architecture that could be split into two components, the encoder and decoder. This architecture introduced parallelization to learning with neural networks. The input embeddings tokenize the original sentences into tokens. Tokens are converted into high-dimension

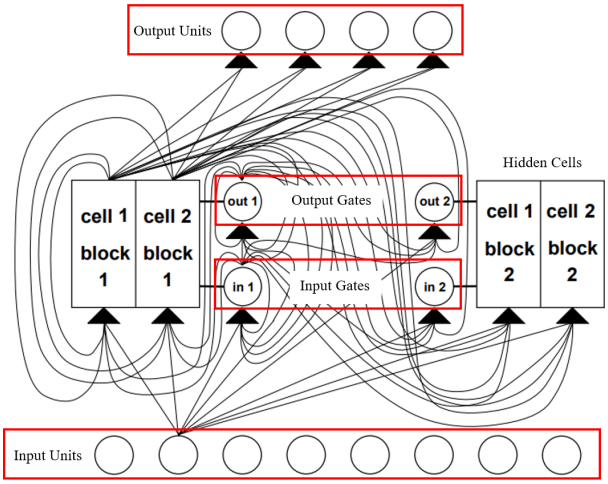

Figure 2.3: Example of LSTM net with 8 input units, 4 output and 2 memory cell blocks. Adapted from [16].

vectors, and then the sequence of the tokens is saved with Positional Encoding (the position of each token is encoded in each vector) [56].

The encoder, which is the left side of Figure 2.4, processes the input sequence derived from the Positional Encoding and represents the first vectors in a matrix. Each layer in the encoder consists of two main sublayers: a multi-head self-attention mechanism and a feed-forward neural network. The self-attention mechanism allows the encoder to focus on different parts of the input sequence dynamically, while the feed-forward network refines the representations. The Multi-Head Attention mechanism, that enables the model to attend different parts of the sequence simultaneously, is composed by attention heads that calculate attention scores. The scores are obtained from the following formula, where queries (Q), keys (K) and values (V) are vectors, while T, called *Temperature*, is a way to control the scale of the scores, which proved to improve models [35, 56]. And d_k is the dimensionality of the keys.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.4}$$

The decoder, represented on the right, generates sequences using predictions from an output shifted right, which means that for a position, it only depends on the known outputs (tokens before the masked token), to predict the next token, and the backpropagation algorithm optimize the weights of the network. The decoder consists of three sublayers: a masked multi-head self-attention mechanism, a multi-head attention mechanism over the encoder's output, and a feed-forward neural network. This structure ensures that the decoder can generate coherent outputs while attending to relevant parts of the input. The input embedding stage is a mix of token embeddings and positional encodings, which are added together to form the final input representation for the encoder. Transformers use attention mechanisms that can process data in parallel, unlike sequential approaches such as RNNs. Modern hardware leverages this parallelization and, with the attention mechanism, this architecture is efficient in handling large datasets, and efficient mod-

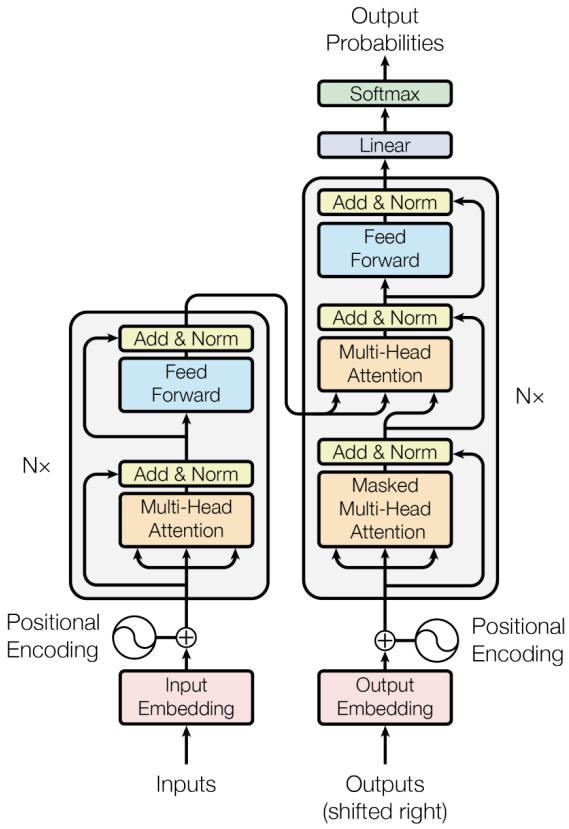

Figure 2.4: Transformers architecture diagram. On the left, the encoder and on the right the decoder. With permission from [56].

els have been produced (e.g., DistilBERT [52]) [37]. In the final step, a softmax function is used to obtain the output probabilities for each token from the "logits" calculated. Seq-to-seq models, such as those used for machine translation or text summarization, leverage both the encoder and decoder sides of this architecture, making Transformers one of the most effective and versatile frameworks in deep learning [56].

Neural networks and other NLP models do not work when an input is text. So, three steps are needed to create input for the models: pre-processing, tokenization, and embeddings creation. To remove some bias in the data that serves as input to a model, some optional steps are relevant, such as stemming – reducing a word to its root form (e.g. "run" is the root form of "running") – and stop-words removal – removing words that do not contribute to the meaning of a text. Tokenizing the input sequence is the process of splitting text into units called tokens. One common way of creating embeddings is using algorithms that represent words or paragraphs in high-dimensional space representations (i.e. large vector representations). One example of an algorithm that creates these representations is doc2vec, a model for learning vector representations of text documents. It is similar to word2vec, which generates multi-dimension vectors to represent words, but instead of words, doc2vec applies the same logic to documents or paragraphs, and creates these vectors to represent documents [41, 29]. This can be applied to capture the semantics of a whole document

(i.e. document embeddings). It creates a vector for each document. The vector represents the semantics of the document, and closer vectors mean they represent documents that are semantically similar, in other words, the same concept that is inside the attention mechanism in transformers. Some useful tasks include document classification, clustering, and other NLP tasks [29].

2.2.2 Sequence Models

Sequence models are probabilistic models that have played a central role in the development of the NLP field. Hidden Markov Model (HMM) laid the foundation for this line of research, later followed by models such as the Maximum-Entropy Markov Model (MEMM) and Conditional Random Field (CRF), each introducing distinct concepts. Both HMM and MEMM rely on the Viterbi algorithm for decoding, which identifies the most probable sequence of hidden states.

HMM is a probabilistic model initially introduced for tasks in speech and signal processing and is applied in analysis of sequential data [4]. It has the particularity of being able to model a process where the system transitions between hidden states, producing observable outputs depending on the states. Figure 2.5 represents a simple HMM, where the circles on the top are the hidden states and the bottom ones are the observations. These observations (X_i) exclusively depend on the corresponding state and the state Y_i only depends on state Y_{i-1}.

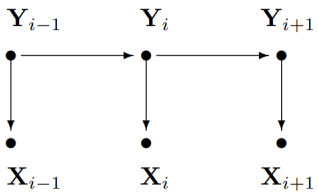

Figure 2.5: Graphical structure of a simple HMM. Adapted from [28].

The main components of an HMM model are hidden states (*S*), observations (*O*), and probabilities, specifically transition probability (*A*), emission probability (*B*), and initial probability (π) and an HMM model could be represented with the following simple notation 2.5.

$$\lambda = (A, B, \pi) \tag{2.5}$$

It should be noted that the model also receives two parameters: N (the number of states of the model) and M (the number of distinct observation symbols per state). Individual states are denoted as $S = \{S_1, S_2, \ldots, S_N\}$, the state at time t as q_t and individual symbols, i.e., the set of possible observations, as $V = \{V_1, V_2, \ldots, V_M\}$. Equations 2.6 to 2.8 define transition, emission and initial state probabilities, respectively. The distribution $A = \{a_{ij}\}$ is defined as the following formula.

$$a_{ij} = P(q_{t+1} = S_j \mid q_t = S_i), \quad 1 \leq i, j \leq N. \tag{2.6}$$

And the distribution $B = b_j(k)$ in state *j*.

$$b_j(k) = P(v_k \text{ at } t \mid q_t = S_j), \quad 1 \leq j \leq N,\ 1 \leq k \leq M. \tag{2.7}$$

The initial state distribution $\pi = \pi_i$

$$\pi_i = P[q_1 = S_i], \quad 1 \leq i \leq N. \tag{2.8}$$

With values for *N*, *M*, *A*, *B*, and π, an HMM model can create a sequence of observations $O = O_1, O_2, \ldots, O_T$. This model also relies on two important assumptions: (a) the *Markov assumption*- a sequence of states $Y_0, Y_1, Y_2, \ldots, Y_N$, the future state Y_{N+1} is only influenced by the last state Y_N; (b) the probability of transition from state *i* to state *j* remains the same throughout the sequence; this is known as the property *time stationary*. In the spectrum of cybersecurity, for instance, HMM is used in Intrusion Detection System (IDS) because it recognizes patterns, and it could predict the hidden state of malicious or benign connections in this domain. An usual HMM model outputs the 3 following matrices, that correspond to the equations: Start, Emission and Transition probabilities matrices. These allow a model to be saved and to be used or be trained over at another time, because being a probabilistic model, with these probabilities, the model can infer the hidden states and do a certain task, such as PoS tagging.

The Viterbi algorithm is a fundamental technique used to find the most probable sequence of hidden states in a HMM, given a sequence of observed events. It is used in sequence models, more specifically in tasks such as Part-of-Speech tagging and speech recognition, where it is used to decode the most likely label sequence.

```
Input:
    O = observed sequence [o_1, o_2, ..., o_T]
    S = set of states [s_1, s_2, ..., s_N]
    pi(s) = initial probability for state s
    a(i,j) = transition probability from s_i to s_j
    b(j,o) = emission probability of observing o in state s_j
Initialisation:
    for each state s in S:
        delta[1][s] = pi(s) * b(s, o_1)
        psi[1][s] = 0
Recursion:
    for t = 2 to T:
        for each state s_j in S:
            delta[t][s_j] = max over s_i in S of [delta[t-1][s_i] * a(
                s_i, s_j)] * b(s_j, o_t)
            psi[t][s_j] = argmax over s_i in S of [delta[t-1][s_i] * a(
                s_i, s_j)]
Termination:
    q_T = argmax over s in S of delta[T][s]
Backtrace:
    for t = T-1 down to 1:
        q_t = psi[t+1][q_{t+1}]
Return the most probable state sequence Q = [q_1, q_2, ..., q_T]
```

Listing 2.4: Viterbi Algorithm (Pseudocode).

The pseudocode shown in Listing 2.4 can be split into four main steps:

1. Initialisation: For each possible hidden state at time $t = 1$, the algorithm calculates the initial probability of being in that state ($\pi(s)$), multiplied by the likelihood of emitting the first

observation ($b(s, o_1)$). These values are stored in the matrix `delta`, which keeps track of the highest probability of any path reaching each state at every time step.

2. Recursion: For each subsequent time step t, and for each state s_j, the algorithm determines the most probable previous state s_i that could transition to s_j, using the recurrence:

$$\delta_t(s_j) = \max_{s_i \in S} [\delta_{t-1}(s_i) \cdot a(s_i, s_j)] \cdot b(s_j, o_t)$$

This step captures the highest scoring path leading into state s_j at time t, taking into account both transition and emission probabilities.

3. Termination: At the final time step T, the algorithm selects the state with the highest probability in `delta[T]` as the end of the most probable path.

4. Backtrace: Using the `psi` matrix, the algorithm traces back from the final state to the beginning, reconstructing the most likely sequence of hidden states.

The algorithm works by recursively computing the highest probability path to each possible hidden state at every time step, storing both the probabilities and back-pointers to reconstruct the optimal sequence. It assumes that the current state depends only on the previous state (the Markov property).

The MEMM is a discriminative model that relies on the Markov assumption that combines the fundamentals of HMM with maximum entropy. For each sequence of observations, the model predicts the most likely sequence of labels by estimating the conditional probability of each state given the current observation and the previous state – this relation is evident in Figure 2.6, the empty circles represent the observations, that are directly known (X_i), and the full circles are the hidden states, more specifically, the labels (Y_i). Unlike generative models such as HMM, which model the joint probability, the MEMM focuses on modeling the conditional probability (Equation 2.9).

$$P(y_i \mid x_i, y_{i-1}) \tag{2.9}$$

Where y_i is the current state, x_i is the current observation, and y_{i-1} is the previous state. MEMM uses maximum entropy, i.e., a framework for estimating probability distributions from data. Maximum entropy is based on the principle that the best model for the data is the one consistent with a set of constraints derived from the training data itself, making the least possible assumptions. This means that it is closest to the uniform distribution, which is the one with the highest entropy following the maximum-likelihood distribution as shown in equation 2.10.

$$P_{s'}(s \mid o) = \frac{1}{Z(o, s')} \exp\left(\sum_a \lambda_a f_a(o, s)\right) \tag{2.10}$$

where s' is the previous state, s the current state, a is a feature, λ_a is the parameters to be learned (the weights), $Z(o, s')$ is the normalizing factor that ensures the distribution sums to one over all possible states s and $f_a(o, s)$ is a feature function which is 1 if a binary feature of an observation is true and the state in time t is equal to the current state, and 0 otherwise. MEMM is also vulnerable

to the *label bias* problem, because the normalization is done in every state, separately, which limits the model capacity to consider the full context, adding bias to the transitions.

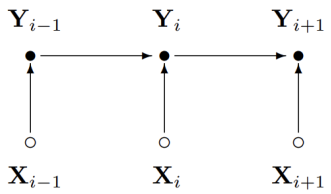

Figure 2.6: Graphical structure of a simple MEMM. Adapted from [28].

The CRF model is a discriminative model that, similarly to MEMM, is trained to calculate the maximum conditional probability given a sequence of states, but it is different because it is able to model features that are not independent and normalize globally through every possible sequence. As shown in Figure 2.7 , the connections between the hidden states (Y_i) do not have arrows because CRF captures the information between adjacent states, not only the last state. This flexibility/robustness allows CRFs to consider overlapping and interdependent features, which can better represent complex relationships in the data. CRF also leverage the use of an *energy function* to measure the compatibility between the observations and the labels. This makes CRF a model that is able to perform tasks such as POS (Part-of-Speech) tagging, data labelling, and other sequence labeling tasks.

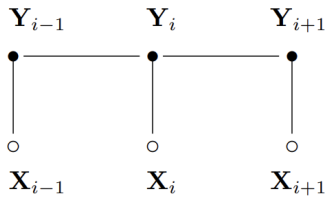

Figure 2.7: Graphical structure of a simple CRF. Adapted from [28].

2.3 Vulnerability Detection Tools

In the last decades, many tools to identify vulnerabilities were developed, and some use intermediate code representations, and then train – machine learning, neural network or sequence – models with these custom corpora. One example that leverages these intermediate code representations and machine learning is MERLIN. MERLIN [11] is a tool developed by Figueiredo et al. with the objective of addressing two problems in static analysis tools: using a specific programming language and the need to write explicit rules to detect vulnerabilities. MERLIN uses machine learning and an intermediate language to find vulnerabilities in applications written in different languages, specifically PHP and Java [11]. The tool works by first converting source code into Java bytecode and then into an intermediate format called Jimple. MERLIN builds control-flow graphs from the code and performs data flow analysis to identify parts of the code that might be

vulnerable. These parts, called code slices, are then analyzed with machine learning to determine if they are safe or unsafe [11].

An example that takes advantage of sequence models is DEKANT, which is a tool that leverages NLP techniques to identify vulnerabilities in web application source code. The DEKANT approach was to use a HMM to process source code similarly to a natural language and then train the model to detect vulnerabilities. In order to leverage this assumption, the source code is translated into an Intermediate Slice Language, which filters irrelevant features and represents the relevant characteristics of the code such as source of input, sanitization functions, validation or sensitive sinks [39]. These translations form sequences of tokens that are used as observations for the HMM. The HMM is trained using a labelled corpus of safe and unsafe code slices, allowing it to learn patterns that identify vulnerabilities. The DEKANT tool was evaluated on 21 PHP applications and 25 WordPress plugins, detecting vulnerabilities in different classes, including SQL Injection, XSS, and others. The tool identified known and zero-day vulnerabilities with high accuracy (approximately 97%) and low false-negative rates, performing better than other tools such as WAP [39, 38]. Outside of vulnerability detection, but in the context of cybersecurity, HMM and CRF have proved to be helpful in pattern recognition tasks, the core of an Intrusion Detection System (IDS) [25, 8].

More recent approaches to vulnerability detection incorporate neural networks. This is the case of SySeVR [32]. It is a framework that converts C/C++ source code, removing some punctuation and other irrelevant characters, and even changes the order of the code slice to create a more suitable input for the model. Other relevant project to mention is WAP [38]. With taint analysis, that creates a set of possible vulnerabilities, and data mining, to predict the false positives, PHP source code is translated to an Abstract Syntax Tree (AST). The data mining component used multiple ML algorithms, graphical, probabilistic and neural network algorithms, such as Random Forest, Logistic Regression or Support Vector Machine, respectively. This static analysis tool also applied code correctness to help developers sanitize inputs, and showed a high accuracy, 92.1% over close to 1.4 million lines of code, finding 388 vulnerabilities. Using LLVM, which allows intermediate code to be executed according to a provided lexer and parser, VulChecker creates a custom compiler to create a Graph Neural Network (GNN)-optimized representation [42].

Fidalgo et al. [10] proposed a model to detect PHP SQLi vulnerabilities with the layers: Embedding, LSTM, Dropout and Dense. This project converted PHP to PHP opcode, an intermediate language representation. 95% accuracy was achieved with the RMSProp optimizer, after fine-tuning the model with various hyperparameters. VulDeePecker is another project that leverage LSTM, more specifically, a Bidiretional LSTM to detect vulnerabilities, upgrading the state-of-the-art [33].

With transformers, models for vulnerability detection have been trained. For source code written in C and C++, VulDeBERT [27] and VulBERTa [15] were developed. VulDeBERT, that uses the encoder part of the transformers architecture, was able to produce better results than those in VulDeePecker also for vulnerability detection and processing the source code by abstracting it

and creating a code gadget that behaves as a natural language [27]. VulDetect [45] uses a pre-trained Generative Pre-trained Transformer (GPT) and achieved better results than VulDeBERT [27].

Overall, despite the progress achieved by tools such as MERLIN, DEKANT, SySeVR, VulDeBERT, VulDetect and others, current approaches still present several limitations. Most tools rely on specific types of models—such as HMMs, LSTMs, or transformer encoders—that, while effective for certain vulnerability classes, struggle when confronted with code patterns that fall outside the distributions seen during training. In addition, many studies report high accuracy but are evaluated on curated datasets or on a narrow subset of vulnerabilities, raising concerns about their ability to generalize to large-scale, heterogeneous, real-world codebases. These tools also often lack the replicable details needed to obtain similar results and provide limited or no explanation of their predictions. In this dissertation, we aim to address these limitations by proposing a multi-model approach that can be extended with an explanatory layer.

Chapter 3

Proposed Solution & Design

In this chapter, we explain the challenges we want to overcome, propose our solution, and explain our design choices. With the challenges described, the architecture we propose to tackle them will be introduced in more detail. We cover the planned steps in using an NLP solution, which involves different NLP models, a translator, and other techniques (e.g., stemming), to detect and explain vulnerabilities in web applications' source code. The chapter starts with the section of challenges (Section 3.1), followed by the architecture of the tool (Section 3.2), called VulNLan, that we want to develop to address the challenges. After explaining the tool's architecture, we dive into our translation methodology (Section 3.3) where we explain in detail the objective of having a translation and how it works. The Training and Testing Phase (Section 3.4) is the next section, where we describe the first steps we are going to take to create our tool and justify our design choices, more particularly, the construction of our datasets and more. The last section of this chapter, named Vulnerability Detection Phase (Section 3.5), describes the second and last stage of our methodology, where we keep discussing our design choices and finish the proposal of our solution.

3.1 Challenges

With the goal of creating an NLP-based tool to detect and explain vulnerabilities in the source code of web applications, we face many challenges. The first comes with our methodology that creates an Intermediate Language (IL) to normalize and simplify the input. To classify and explain the vulnerabilities, we use multiple NLP models that receive different inputs. With our different models, we want not only to classify the files as vulnerable or not vulnerable but also explain these classifications.

Challenge 1: Ensure the logic of the code when representing PHP in our IL.

A common scripting language for web applications, such as PHP, has a large and complex language specification. Every feature of the PHP language needs to be processed when the translation occurs. If the logic is not maintained, the vulnerability detection will not be effective. This challenge is complex because we want to remove bias-inducing elements while keeping the logic

of the original file. To keep this logic we have the obligation of maintaining code structures when translating, solely removing the tokens that do not contribute to the logic of the code.

Challenge 2: Creating datasets for multiple models.

NLP models receive different types of inputs depending on the architecture of each model and the task that the model will be trained for. One example is the difference in inputs between a supervised Hidden Markov Model, which has the goal of finding a hidden state, given its training with hidden state annotated files, and a neural network that learns patterns with file tokens. Therefore, it is required to create different datasets representative from the same original dataset, but follow the models requirements and ensure the same reliability on data they contain.

Challenge 3: Explain the classification results.

Explaining the results of our classification models adds a valuable layer to our approach but it is one more problem that we need to solve. We will need to explain the results, and keep the methodology of using NLP techniques, now with this explanatory task. To overcome this, we need to find a methodology that allow us to obtain the taintedness propagation token by token.

3.2 VulNLan tool architecture

Our solution to tackle the challenges described in the previous section, with the main objective of detecting and explaining vulnerabilities in the source code of web applications, will be introduced and discussed on this section. We propose a tool, named **VulNLan**, that is able to receive web application code snippets, more specifically PHP code, translate them into our IL and, with multiple models, find and explain vulnerabilities in those snippets. To achieve this, we created an architecture to develop the tool, showcased in Figure 3.1, that has two main phases: *Training and Testing Phase* and *Vulnerability Detection Phase*. This architecture is structured as a complete system, where data flows from the input sources through multiple processing and learning components until the final interpretation and explanation of vulnerabilities.

The workflow begins with the PHP dataset, which is processed and translated into our intermediate language (IL). Processing and translating converts the original PHP code to our IL with the goal of reducing bias in the training corpus to improve the performance of the models. From this step, distinct datasets are produced. These datasets will be constructed to be effective in the tasks we want our models to excel, either classification and explanation tasks.

We use distinct model families because supervised sequence models are well-suited for identifying hidden states in token sequences, making them effective for explanation, while neural models excel at recognizing complex patterns, making them more effective for classification. We also use an unsupervised at token-level sequence model to learn the patterns of vulnerable and not vulnerable files to improve our classification task. The IL dataset is the one for the classification, while the annotated dataset is used to train supervised sequence models. Therefore, during this phase,

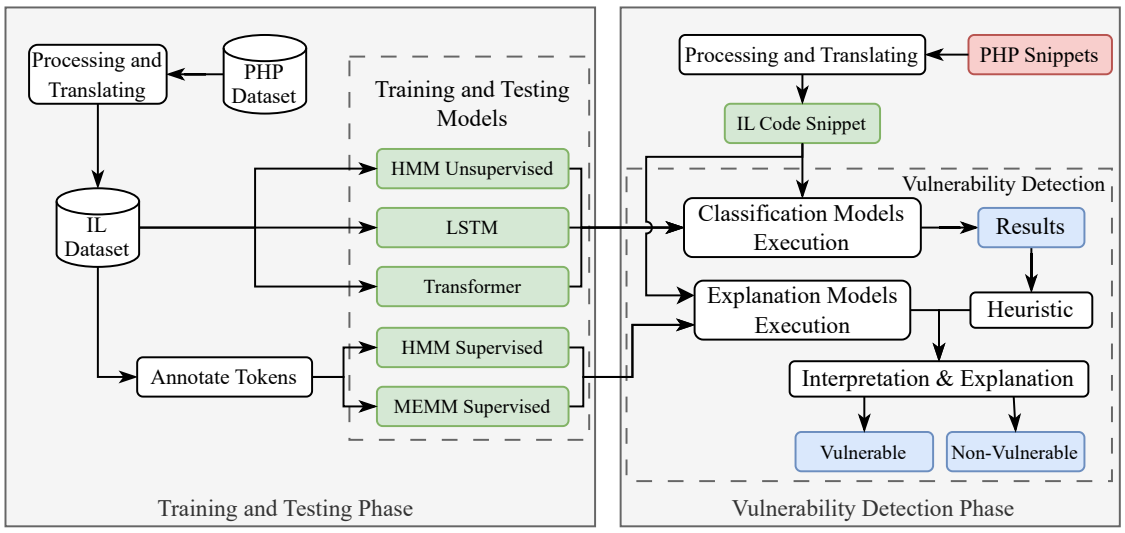

Figure 3.1: Architecture of the VulNLan tool.

a set of unsupervised and supervised datasets are constructed and respecting the requirements of the models. For example, for the supervised sequence model datasets the annotated dataset is used because raw datasets without hidden states are not able to train these models.

In the Vulnerability Detection Phase, a PHP code snippet that we want to check if it is vulnerable follows the same initial steps: it is processed and translated into IL, then passed through the classification models. The results are refined by a heuristic layer and further analysed by the explanatory models. Finally, the system outputs the results of the classification models, marking the snippet as either vulnerable or non-vulnerable, with an accompanying explanation of the result, that labels every token in the file, showcasing the input sanitization or lack of it.

3.3 Translating PHP to Intermediate Language

The key component of our solution, and that belongs in both phases of our approach in Processing and Translating box, is our translator. This element converts the original web application PHP source code into our IL. Translating PHP code to IL is necessary because PHP code is not normalized and the number of tokens is excessively large for some of our models (i.e., for sequence models, we want to avoid never-seen tokens). This translation will also be used two times in our solution: first is before training the models, namely in the dataset preprocessing stage, and the other is before a new snippet is inputted into a model for prediction. Both the dataset and new snippets will be represented in our IL, ensuring the prediction is coherent.

To translate PHP source code into our IL, we need to ensure that elements from PHP are maintained in our IL. The most important aspects of the PHP code to remain consistent are the logical and data flows of the code. Other aspects must also be preserved, such as the SQL query. Without this data, detection of the SQL injection vulnerability would not occur reliably, because the models would not be able to learn the correct patterns in the code.

To reduce the bias present in the PHP code, we decided to change or remove elements. These bias-inducing or unnecessary elements we chose to change or remove are:

- In-line and multi-line comments;
- Static strings, including HTML fragments;
- Opening and closing PHP tags;
- Punctuation;
- Variable names;
- Function and class names;
- SQL queries;
- Control structures.

Comments add bias to the code without changing the data and logical flows; static strings are mainly irrelevant to our tasks (e.g., database names); opening and closing tags are unnecessary tokens; punctuation is irrelevant (e.g., in the end of each line of code, the presence of `;` alters the code functionality but not the existence of vulnerabilities); variable names can add a large number of different tokens to our corpus, so we decided to normalize them, as with function and class names; SQL queries are abstracted since different query structures do not affect vulnerability detection; control structures such as `while` or `for` loops will be normalized and simplified to reduce bias in our dataset.

These changes aim to simplify model pattern recognition and reduce the vocabulary of the training corpus, which can improve the results, especially for sequence models.

We chose an IL instead of raw PHP code for the reasons outlined above, and because we believe that using an IL without all language-specific syntax can improve vulnerability detection.

3.4 Training and Testing Phase

Sequence and neural network models are prone to overfitting on the training data. To address this, our approach focuses on efficient training while making full use of the available data. We leverage the specific strengths of each model type and follow established practices for training and testing under controlled conditions. These objectives form the basis of our translation and dataset preprocessing strategies. In this training and testing phase, we present the baselines of our translation methodology, followed by a dataset review that provides a detailed description of the data and outlines our plan for training the classification and explanation models.

3.4.1 Dataset

The dataset that will be used in our models is the Software Assurance Reference Dataset (SARD) from NIST [44]. This dataset is focused on the Common Weakness Enumeration (CWE)-89 vulnerability, SQLi. It is composed of safe and unsafe files, each one of them with one of the two

labels (safe and unsafe). This synthetic dataset is entirely composed of PHP files, more specifically, a total of 1,362 files – 860 files vulnerable to SQLi and 502 safe files – making this dataset suitable for our solution.

As mentioned above, the dataset is focused on SQLi, but it also contains the XSS reflected vulnerability because when an input is not sanitized and reaches an `echo` function (a function sensitive to XSS), it will be rendered by the web application.

The dataset combines multiple aspects of the PHP language, from simple files to more complex ones with functions and classes. This diversity is beneficial for the training of our models. Another positive aspect of this dataset is that each file is representative of code that could appear in real-world web applications where SQLi is present.

The files from the dataset will be preprocessed and translated to IL before the training. The dataset necessary for the supervised sequence models will go through another step, i.e., the annotation step. The annotation will follow a predefined set of hidden states: `taint`, `ntaint`, `san` and `und`. Each state has a specific function that tracks the data flow. `taint` is the state in which tokens have a tainted payload, i.e., can be compromised, such as user inputs from `$_GET` in PHP or any variable assigned to that payload. `san` state represents the sanitization functions, i.e., that invalidate malicious inputs (e.g., the `mysql_real_escape_string` token that appears widely in the dataset). `ntaint` state is for tokens that receive sanitized content. And lastly, the `und` state is for tokens that do not contribute to the taintedness of the data flow (e.g., `class`, `if`, and more). The annotation will be made and verified manually.

As mentioned, the dataset is not balanced, so, in order to use it in the training, it must be balanced. To achieve this we can use 500 vulnerable and 500 not vulnerable files, but choosing files of every category to show the models the best range possible with the dataset.

This dataset is well-suited for both classification and explanation tasks: for classification, it provides file-level vulnerability labels; for explanation, once token-level annotations are added, it will contain sufficient patterns for token hidden state labelling (i.e., the basis of our explanation).

3.4.2 Models training

We aim to train multiple models: HMM, MEMM, LSTM and Transformers. HMM will have two separate methodologies, one HMM unsupervised will work as a classifier while HMM supervised has the task of labelling each token of the code snippet in analysis, thus providing and explanation regarding their taintedness. MEMM will solely be focused on the explanatory task with its supervised training. And, lastly, LSTM and Transformers models will be focused on the classification task.

The dataset used for model training is the IL dataset. For the unsupervised training we use the IL dataset with no other changes. In the supervised training for the explanation task we use the IL annotated dataset.

In the unsupervised HMM training, that, unlike the supervised one, is focussed on the classification task, we train two models and check for which model the instance in analysis has better

probabilities, thus determining the classification for this instance using this type of model.

LSTM and Transformers models training will occur with the IL dataset with hyperparameters fine-tuned for each model. In the LSTM, we will use the best optimizer over the three we will test: `Adam`, `Adadelta` and `RMSProp`. The process for Transformers model will be similar, but instead of optimizers, we will test with different pre-trained models, namely `CodeBERT`, `GraphCodeBERT`, `RoBERTa` and `GPT-2`. Pattern recognition in these deep learning architectures handles our classification task. But, in order to achieve the best possible results with our dataset, we decided that our approach includes a cycle of training, testing and fine-tuning. At each iteration, each model will train with certain hyperparameter values, which will be tuned between cycle iterations, until achieving satisfactory results. This fine-tuning process is represented in Figure 3.2.

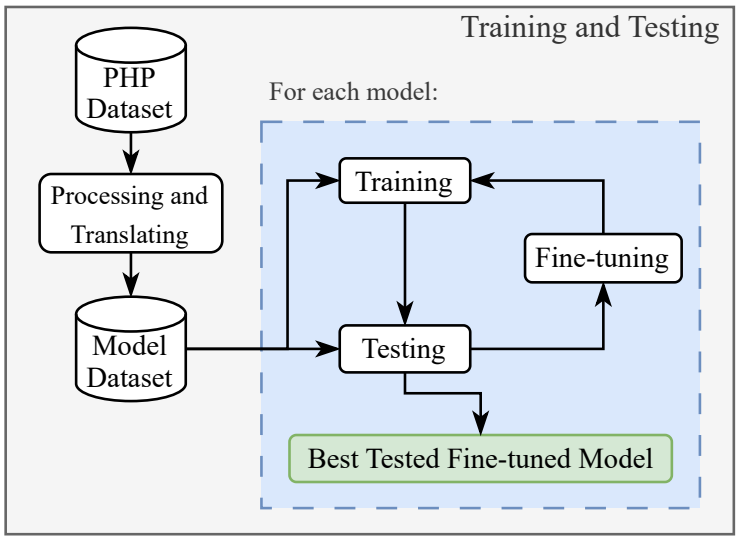

Figure 3.2: Model development cycle, including training, testing, and fine-tuning.

In the HMM supervised training we will train two separate models, one with the vulnerable instances of the dataset, and other with the safe files. This way, each model will understand safe and unsafe patterns, and will be able to label each token accordingly to the hidden state. MEMM training will be similar to HMM supervised using the full annotated dataset. The probabilistic base in these models is adapted for the labelling token by token that occurs in our explanatory approach. With this methodology we can show the taintedness propagation or sanitization on the file tokens, so it will be clear if it reaches a sensitive sink. Even if this sequential models do not have clear hyperparameters, they will follow the development cycle presented in Figure 3.2, where some tweaks in the token labelling can improve the overall efficacy of the model.

In total, we are going to use two different datasets for our training: IL dataset serves as an input for classification models, and it is not supervised at token-level; For explanation we use the labelled token by token annotated dataset, that, unlike the IL dataset, is supervised at token-level.

3.5 Vulnerability Detection Phase

In this section, which represents the second and last stage of our approach, we have the goal to connect the elements in our architecture to form our vulnerability detection and explanation tool (i.e., detect the presence of vulnerabilities in a web application source code, and provide an explanation to those results). To make this a proper and effective vulnerability detection, we leverage the utilization of the neural network models plus the HMM unsupervised probabilistic model. To determine the final classification, and therefore join the models classification in a single one, we propose an heuristic that uses the models accuracy as being their respective weight to calculate the final probability of a code snippet being vulnerable or not vulnerable. Meanwhile, the explanation is provided solely by the probabilistic models, in this case, supervised HMM and MEMM. These models together with our translator and heuristic will form our tool, that we call VulNLan (an acronym for **Vul**nerability detection with **N**atural **Lan**guage).

3.5.1 Heuristic

When using multiple models for classification, an important design challenge arises: models may disagree on the label of a given snippet. Since our system requires a single final prediction, we needed to design a heuristic that combines the output of all models into one decision.

Our approach uses the probabilities produced by each model (HMM, LSTM, and Transformer) and weight them according to the accuracy of the respective model. This way, more reliable models have a greater influence on the final classification. Joining the accuracy of the models to the classification result, we can calculate this weighted score. The best performing models will have a bigger contribution to our confidence score than the low performing ones. Specifically, as each model outputs two probabilities – the not vulnerable (NV) and the vulnerable (V) – for the code snippet, the heuristic calculates for each classification its final probability, considering the accuracies and probabilities of the models given for each classification. These final probabilities we denominated by confidence scores.

Thus, the confidence scores for each class are defined as:

$$C_{NV} = acc_{HMM} \cdot P_{HMM}(NV) + acc_{Transf} \cdot P_{Transf}(NV) + acc_{LSTM} \cdot P_{LSTM}(NV) \quad (3.1)$$

$$C_{V} = acc_{HMM} \cdot P_{HMM}(V) + acc_{Transf} \cdot P_{Transf}(V) + acc_{LSTM} \cdot P_{LSTM}(V) \quad (3.2)$$

The final classification is obtained by selecting the class with the highest confidence score between the results of Equations 3.1 and 3.2. To improve interpretability, the raw confidence values (which lie in $[0, 3]$ because each accuracy and probability values stand between $[0, 1]$) are rescaled into percentages (i.e., value domain of $[0, 100]$), providing an intuitive measure of how strongly the ensemble supports the chosen classification. With confidence values intervals, we define our confidence model with the Table 3.1.

The class chosen for the explanation models is the class with the greater confidence value. If the results were close to 50, for both vulnerable and not vulnerable classes, the confidence level

Table 3.1: Confidence intervals and corresponding confidence levels.

Interval (%)	**Confidence Level**
0–20	Very High
21–40	High
41–59	Low
60–79	High
80–100	Very High

is marked as Low. The sum of each model vulnerable and not vulnerable probabilities equals to 100%. The confidence is Low if the probabilities for each class are close to each other, and High or Very High if the distance between them is greater. For example, a snippet, after being classified by the models, having a confidence value for vulnerable (C_V) of 85, it fits in the very high confidence level, which is the last line of Table 3.1. If in this same snippet C_{NV} is, for example, 15, it also has a very high confidence level that the snippet it is not non-vulnerable, and we can confidently mark the snippet as vulnerable.

To distinguish more certainly the class, we add a decision table where we check the class that the majority of models has chosen. This supports the classification to be used on the explanation models. For example, to use the vulnerable explanation models (i.e., the supervised HMM and MEMM that were trained with annotated vulnerable files), the tool chooses the major class between the 3 classification models, as we can see on Table 3.2.

Table 3.2: Decision table by classification models output (3 models).

HMM	**Transformers**	**LSTM**	**Majority class**	**Final class**
NV	NV	NV	NV	NV
NV	NV	V	NV	NV
NV	V	NV	NV	NV
V	NV	NV	NV	NV
V	V	NV	V	V
V	NV	V	V	V
NV	V	V	V	V
V	V	V	V	V

Even if the confidence value for C_V is higher than 50, the explanation models are going to use the not vulnerable models if two models classified the snippet as not vulnerable.

This design allows us to combine the strengths of all models, while remaining flexible to adapt if new models are added or their performance changes in the future (e.g., using better pre-trained Transformers models, or finding better parameters).

Chapter 4

Implementation

This chapter presents the implementation of the proposed solution, detailing how the conceptual design was transformed into a functional system. While the previous chapter outlined the architecture and design decisions of VulNLan, here the focus is on the practical realization of its components. The implementation covers the development of the translator to the intermediate language (IL), the construction and preprocessing of datasets, the training and fine-tuning of classification and explanation models, and the integration of these components into the VulNLan tool. The final stage concerns the classification and interpretation of code snippets, where the system determines whether a snippet is vulnerable or not.

The chapter is organized as follows. Section 4.1 describes the development of the translator. Section 4.2 explains the construction of the datasets. Section 4.3 details the training process for classification and explanation models. Section 4.4 presents the integration of all components into the VulNLan tool. Finally, Section 4.5 describes the classification and interpretation stage.

4.1 IL Translator

The first step in VulNLan development that follows the architecture described in Figure 3.1 is to create the translator that converts PHP source code into our IL code, which we called PHP2IL. This will be useful before training the models and right before executing the trained models, for dataset translation and PHP code snippet translation, respectively. Also, we distinguish this translator from a pre-processing stage because we not only clean the PHP code, but also create a new language (i.e., IL) by changing tokens and creating rules. The corpus of our models will be composed of similar files to the PHP file represented in Listing 4.1.

At first, we developed this translator as a function just to process a dataset. Then we realized that creating this dataset as a tool would be more practical and efficient because this helps project replication and can be executed on a command line interface, where the two arguments are the source folder or files and the destination is the IL files target folder. At this beginning stage, we were using a Comma-Separated Values (CSV) composed of PHP tokens and each token's respective label. This was not close to being the most effective approach. Later, we decided to handle each aspect of a PHP file by parsing.

```
<?php
/*
Unsafe sample
input : Uses popen to read the file /tmp/tainted.txt using
    cat command
sanitize : none
construction : interpretation
*/
$handle = popen('/bin/cat /tmp/tainted.txt', 'r');
$tainted = fread($handle, 4096);
pclose($handle);
$tainted = mysql_real_escape_string($tainted);
$query = "SELECT * FROM COURSE c WHERE c.id IN (SELECT
    idcourse FROM REGISTRATION WHERE idstudent= $tainted )
    ";
//flaw
$conn = mysql_connect('localhost', 'mysql_user', '
    mysql_password'); // Connection to the database (
    address, user, password)
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;
$res = mysql_query($query); //execution
while($data =mysql_fetch_array($res)){
print_r($data) ;
echo "<br />" ;
}
mysql_close($conn);
?>
```

Listing 4.1: Example of PHP file from the dataset we will use as corpus for the models.

With the decision of parsing PHP code, by elements in the code (i.e., loop cycles, conditional statements, classes, and more) we created a class, called `PHPDecomposer`, where we save each relevant aspect of each PHP file: the file content, the name and relative directory of the file, and the classification label in the case of being a supervised dataset.

This class, with its constructor and methods presents a clean way to create a parser, step by step. The first step we choose to do was to remove the elements that induce the most bias in the PHP files, i.e., the comments. After researching the types of comments on PHP code, we found four types of comments to remove: inline comments, multiline comments, `heredoc` and `nowdoc`. After developing these functions with regular expressions (i.e., a sequence of characters that create a search pattern for strings), we still had many steps in front of us to create our IL.

The next steps in the PHP2IL dataflow involved progressively refining the representation of PHP code. In addition to removing comments, we also removed static strings, which often introduce bias since they may contain tokens similar to PHP code. SQL queries were also normalized: the fixed part of the query was replaced with a standard `SQL_QUERY` token, followed by the dynamic part (typically variables). This ensures that the query structure is preserved without leaking dataset-specific identifiers.

Control flow structures such as `if`, `else`, `for`, `while`, and `do-while` were preserved

```
1
var1 popen
var2 fread var1 4096
pclose var1
var3 SQL_QUERY var2
var4 mysql_connect
mysql_select_db
echo var3
var5 mysql_query var3
loop var6 mysql_fetch_array var5
  print_r var6
  echo
mysql_close var4
```

Listing 4.2: Example of IL file translated from PHP file in Listing 4.1 after code obfuscation.

in their original form to maintain the program's logical flow. To ensure that indentation was respected, a counter was maintained to track the level of nesting, reflecting the internal logic of the code. Similarly, array declarations were regularized, and PHP superglobals (e.g., `$_GET`) were normalized to more generic tokens (e.g., `GET`).

We also removed punctuation whenever possible, relying instead on tokens and indentation to capture the underlying program logic. This decision contributed to reducing noise and focusing the IL representation on the essential semantic aspects of the code.

One of the most impactful improvements came from code obfuscation. After several iterations of testing the dataset with Transformers models, we observed that normalizing tokens improved the quality of the results. Such tokens, specifically the names of variables, classes, and functions were replaced with standardized names: `varX`, `classX`, and `functionX`, where `X` corresponds to the order in which the element appears in the code. This step ensured that the IL representation remained faithful to the original program structure while avoiding overfitting to specific identifiers. In Listing 4.2, we present the translation from the PHP snippet shown before in Listing 4.1.

After finishing the core of our translator, we decided to also create a dataset with files similar to Listing 4.2, but removing variable Identifiers (IDs). For example, the representation of `var3`, in line 8, it is only represented by `var`. With this new dataset, we want to inquire how models behave with and without IDs and see if it improves or worsens the vulnerability detection capabilities of the models.

At the final stage of the translator dataflow, the processed code is written into files with the `.il` extension. These files are stored in the destination path specified in the program arguments. Once translated, the files are consistent and ready to serve as input for either supervised or unsupervised models. Since the dataset used in this work is labelled at the file level, each IL file from the labelled corpus begins with a label: `0` for safe files and `1` for vulnerable ones. For example, the file of Listing 4.2 begins with 1, meaning that it correspond a unsafe file. However, those PHP files that do not contain any label will not have one in our IL. This design choice ensures that the intermediate language representation is directly aligned with the requirements of the training

phase.

4.2 Datasets construction

To build the datasets that serve as corpus for our models we have to choose a dataset. The dataset chosen was the SARD dataset for injection vulnerabilities from NIST, in this specific case, SQLi vulnerability. This dataset is composed of labelled PHP files not evenly distributed, as represented in Table 4.1.

Table 4.1: Number of instances for injection vulnerabilities.

File Label	**Number of instances**
Vulnerable	860
Not Vulnerable	502
Total	1,362

To check if this dataset was suitable for our task, we need to study the pattern similarity in the dataset with the goal of measuring and understand how much they are diverse in terms of representing different code situations and contexts. We need this because we can get better results if the models train on a wider range of token patterns. To do this, we calculated the similarity between all the files of the translated dataset. Specifically, we calculated the similarity of two versions of the translated IL dataset: a version with obfuscated identifiers and a version without identifiers (i.e., similar to Listing 4.2 but without the number on variable, classes, and functions defined in the code snippet, for example, `var1` becomes `var`).

To quantitatively assess the diversity of token patterns in the datasets, each file was represented as a vector using the TF-IDF scheme, previously defined in Section 2.2. This representation ensures that tokens that occur frequently in a document but are rare across the corpus receive higher weights, while common tokens such as separators or keywords have lower influence. As a result, each file is encoded as a high-dimensional vector that captures the relative importance of its tokens.

Once the TF-IDF vectors were obtained, the paired similarity between all files was computed using cosine similarity. Given two TF-IDF vectors $\vec{v}_i$ and $\vec{v}_j$, their similarity is defined as Equation 4.1:

$$\text{sim}(i,j) = \cos(\theta) = \frac{\vec{v}_i \cdot \vec{v}_j}{\|\vec{v}_i\| \, \|\vec{v}_j\|}, \tag{4.1}$$

where $\vec{v}_i \cdot \vec{v}_j$ is the dot product and $\|\vec{v}_i\|$ is the Euclidean norm of vector $\vec{v}_i$. The cosine similarity ranges between 0 and 1, where values close to 1 indicate high similarity in token distributions between the two files, while values closer to 0 indicate low similarity. After calculating the similarity between file pairs, we obtained the results shown in Table 4.2.

In Table 4.2 we denote a significant difference between the two datasets. The dataset with IDs exhibits a lower mean similarity and a wider spread of values, indicating greater diversity of token

Table 4.2: Statistical summary of pairwise cosine similarities for both datasets.

Metric	With ID	Without ID
Mean	0.4654	0.8310
Median	0.4421	0.8303
Std. Dev	0.1858	0.0963
Minimum	0.1130	0.5768
Maximum	1.0000	1.0000

patterns. In contrast, the dataset without IDs shows higher average similarity and lower variance, which highlights the redundancy introduced when variable identifiers are removed.

In the histogram presented in Figure 4.1, the differences between the two datasets become even clearer. The dataset with identifiers (shown in green) displays a broader distribution of similarity values compared to the dataset without identifiers (shown in red). This wider spread is expected and theoretically more advantageous for training, as it exposes the model to a larger variety of token patterns. Nevertheless, both datasets remain valuable in our study, since they enable an analysis of how models behave under different levels of redundancy.

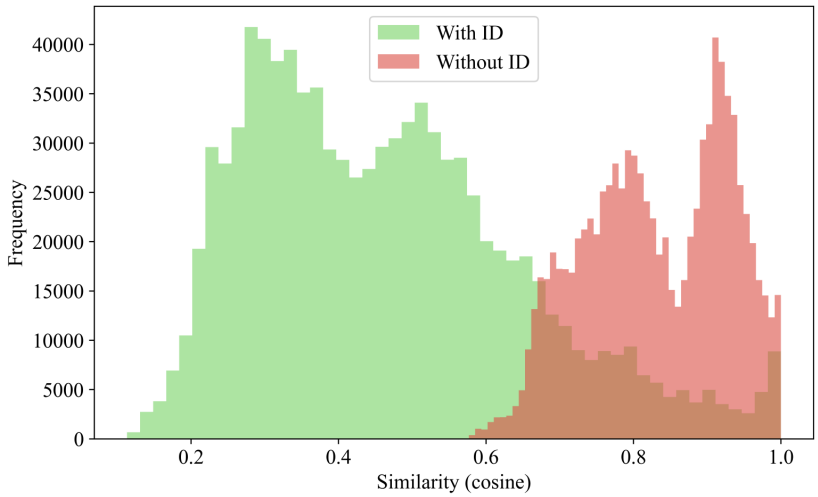

Figure 4.1: Distribution of pairwise file similarities in the two datasets: with identifiers vs. without identifiers.

This procedure provides a straightforward way to assess the structural redundancy of the datasets. A higher overall similarity indicates that many files share nearly identical token patterns, thereby reducing the effective diversity of the dataset. In contrast, a lower average similarity reflects greater variability in code patterns, which can influence the performance of models trained to generalize across diverse inputs.

4.2.1 Datasets for classification models

With the objective of creating the best conditions for our models training, we balanced both versions of the dataset (i.e., with and without identifiers) as seen in Table 4.3.

Table 4.3: Dataset split for training and testing where each class is balanced.

Set	**Vulnerable**	**Non-vulnerable**
Training	450	450
Testing	50	50
Total	500	500

For the classification tasks, the translated IL datasets were adapted according to the requirements of each model and format compatibility with the libraries used while creating the models:

- **Unsupervised HMM model:** the unsupervised HMMs were trained exclusively on the dataset with identifiers preserved, since the distinct variable names provide better sequential patterns for probabilistic modeling.
- **Transformers models:** both dataset versions were used. This dual setup allows us to evaluate the impact of code obfuscation on model performance.
- **LSTM model:** we created CSV (Comma-Separated Values) files because it was more compatible with the libraries used. We choose the CSV representations of the full dataset with identifiers.

The unsupervised HMM and LSTM models dataset chosen was the one with identifiers. The dataset without identifiers did not produce acceptable results, and these were removed from the tool, saving the models trained with identifiers. In the case of Transformers, the performance was similar, so we kept all the models trained.

4.2.2 Datasets for explanation models

The creation of datasets for the explanation models differs from the one of the classification models. While the classification models are trained using IL files at the file level, the explanation models require IL files with their tokens annotated. To achieve this, we manually assigned a state to every token in the sequence, following a predefined set of states (`taint`, `ntaint`, `san` and `und`) as explained in Section 3.4.1. We also split the dataset for training and testing. For testing we used 15% of the dataset for vulnerable and not vulnerable class, as represented in Table 4.4. Listing 4.3 shows, as an example, the file of Listing 4.2 annotated with the states. To annotate the files we also removed lines after the sensitive sink because it was similar in every file, and would not contribute positively to the explanation (i.e., we focused on the taintedness propagation until the sensitive sink, where the input could be sanitized).

Table 4.4: Dataset split for training and testing with annotated files.

Set	**Vulnerable**	**Non-vulnerable**
Training	204	204
Testing	36	36
Total	240	240

```
var1::taint popen::taint
var2::taint fread::taint var1::taint 4096::und
pclose::und var1::taint
var3::taint SQL_QUERY::taint var2::taint
var4::taint mysql_connect::und
mysql_select_db::und
echo::und var3::taint
var5::taint mysql_query::taint var3::taint
loop::taint var6::taint mysql_fetch_array::taint var5::
    taint
print_r::taint var6::taint
echo::und
mysql_close::taint var4::taint
```

Listing 4.3: Example of annotated file from IL file in Listing 4.2 after removing identifiers.

To train the explanation models, we created separate annotated datasets for both vulnerable and non-vulnerable files. Since the classification task is performed first, the explanation models can be selected according to the classification result. Therefore, for both supervised HMM and MEMM, we trained two models: one specialized in vulnerable files and another specialized in non-vulnerable files.

- **Supervised HMM models:** each of the two models learns token-level hidden state transitions specific to its class, enabling them to highlight vulnerable or safe code patterns when explaining predictions.

- **MEMM models:** similarly, two MEMM models were trained, one for vulnerable and one for non-vulnerable files. These models predict the most likely label sequence for the tokens, conditioned on the previous state and current observation, which allows them to provide fine-grained token-level explanations.

Table 4.5: Summary of datasets used for each family of models.

Model type	Dataset	Instances	per class	# of models	ID
Transformers	IL	1000	500	8	Yes and No[1]
LSTM	IL	1000	500	3[2]	Yes
HMM unsupervised	IL	1000	500	2	Yes
HMM supervised	Annotated[3]	480	240	2	No
MEMM	Annotated	480	240	2	No

Table 4.5 provides an overview of the datasets and experimental settings, serving as a reference for the training procedures described in the following sections.

[1] Two datasets were used, one with identifiers and one without, for each pre-trained Transformers model (`CodeBERT`, `GraphCodeBERT`, `RoBERTa`, and `GPT-2`).

[2] Optimizers used: `Adam`, `Adadelta`, and `RMSProp`

[3] Both annotated models were trained twice: vulnerable and not vulnerable datasets.

4.3 Training and fine-tuning models

This section describes the training and fine-tuning procedures of the models used in this work. Two complementary approaches were explored for the classification task: neural network-based models (LSTMs and Transformers) and probabilistic models (unsupervised HMM). The goal of both families of models is to predict whether a given PHP file, represented in IL, is vulnerable or not. While neural networks rely on large-scale representation learning and transfer learning, HMMs exploit the probabilistic structure of token sequences to distinguish vulnerable from non-vulnerable files.

In addition to classification, a second layer of modeling was dedicated to explanations. The explanation models aim to provide token-level interpretations of vulnerabilities within IL files. For this purpose, an annotated dataset was created, and two families of sequence models were trained: supervised HMM and MEMM. Unlike the classifiers, these models do not decide whether a file is vulnerable or not – instead, they assume the classification result as input and focus on revealing which parts of the code contributed most to that decision.

To train each model, we adopted an iterative process of training, testing, and fine-tuning as displayed in Figure 3.2. Each model received its respective dataset and was trained under a controlled setting, followed by evaluation on validation and test splits. Whenever the performance metrics did not meet the desired levels, hyperparameters or preprocessing steps were adjusted, and the cycle was repeated. This ensured that both the classification and explanation models were optimized for the characteristics of the IL datasets.

In the early stages of training the models, this fine-tuning not only changed hyperparameters but also changed details of the dataset that we believed that was degrading the model.

4.3.1 Classification models

As explained before in Chapter 3, we train models to classify snippets in IL. To do this task we are going to train 3 models that contribute in parallel to our final classification: Transformers, LSTM, and unsupervised HMM.

Unsupervised HMM

To train the Unsupervised HMM, we use the dataset separated by the file-level classes (i.e., vulnerable or non-vulnerable). This process is performed for both dataset variants, with and without identifiers. After loading and shuffling the dataset for both classes, the sequences are encoded into numerical form, since the HMM implementation requires discrete numerical observations.

We employ the `CategoricalHMM` from the `hmmlearn` library, configured with 2 hidden states, which correspond to our file-level classes. Each model is trained for 100 iterations. To address the issue of tokens with zero transition probability, we apply *Laplace smoothing*: a small probability mass, controlled by the hyperparameter `alpha`, is distributed over all transitions and emissions, ensuring that no event is assigned a zero probability. Thus, the main hyperparameters

of the model are the number of training iterations and the smoothing factor `alpha`. At this stage, each HMM learns the probabilistic structure of the class it is associated with.

For classification, we train two distinct HMMs: one using only vulnerable files and another using only non-vulnerable files. A new snippet is translated into its encoded token sequence and evaluated against both models. The log-likelihood of the sequence is computed under each model, and the final classification is determined by the model that assigns the higher probability to the sequence. In addition to the predicted label, this process provides a confidence score, since we retain the normalized probabilities from both models. Such probability-based outputs are especially valuable for the heuristic layer, where the relative confidence of different classifiers is combined.

During training, we also store the three main probability matrices of each HMM: the initial state distribution (π), the transition probability matrix (A), and the emission probability matrix (B). These matrices not only define the model's learned structure but also allow for deeper inspection of how vulnerable and non-vulnerable code differ in their probabilistic patterns.

LSTM

For the LSTM classifier, the same IL dataset was used, separated into training, validation, and testing sets. Each file was tokenized and encoded into integer sequences through a vocabulary built from the training set, limited to the 5000 most frequent tokens. All sequences were padded or truncated to a maximum length of 77 tokens, ensuring fixed-size input to the model.

The architecture of the classifier consists of an embedding layer, followed by a stacked LSTM with 5 layers and a hidden dimension of 64 units. A dropout layer with a rate of 0.5 was included to reduce overfitting. The final output is obtained by passing the last hidden state through a fully connected layer with two output units, corresponding to the vulnerable and non-vulnerable classes. The model was trained using the cross-entropy loss, which is well-suited for classification tasks.

To optimize the training process, we experimented with three different LSTM optimizers: `Adadelta`, `RMSprop`, and `Adam`. After comparing results, `Adam` was selected, as it consistently provided more stable convergence and higher validation performance for this task. The learning rate was set to 0.003, and the model was trained for 200 epochs with a batch size of 77. These hyperparameters were iteratively adjusted during development until stable and satisfactory training dynamics were achieved.

During training, model performance was monitored on the validation set after each epoch. The evaluation included loss, accuracy, and precision, ensuring that the model did not overfit and that it generalized well to unseen data. Once trained, the model was tested on the held-out dataset, and the metrics reported included accuracy, precision, recall, F1-score, as well as the confusion matrix.

Finally, in addition to the predicted class labels, we also retained the raw logits and softmax probabilities output by the model for each class. This provided not only a binary decision but also a confidence score for the predictions, which was later integrated into the heuristic layer to combine results from different classifiers.

Transformers

For the Transformer-based approaches we explored a set of pre-trained models from the Hugging Face library, more specifically `CodeBERT`, `GPT-2`, `RoBERTa` and `GraphCodeBERT`. These models were adapted to our classification task through fine-tuning. The dataset was split into training, validation, and testing sets, and all experiments were performed both with and without identifiers in the IL representation.

The tokenization process was performed using the `CodeBERT` tokenizer, which is tailored for source code and is compatible with a wide range of programming constructs. Padding was set to the end-of-sequence token, and truncation was applied to ensure the sequences fit within the model's maximum input length. For the model architecture, we employed a classification head on top of the pre-trained Transformer, predicting one of the two classes: vulnerable or non-vulnerable.

The fine-tuning process was performed using the Hugging Face `Trainer` API. Training was conducted on Graphics Processing Unit (GPU) when available, with typical configurations including a batch size of 8, a learning rate of 1×10^{-5}, weight decay of 0.001, and up to 4 epochs. During training, the model was periodically evaluated on the validation set, tracking metrics such as accuracy, precision, recall, and F1-score. The training and validation losses were also recorded to ensure stable convergence and avoid overfitting.

In addition to the standard fine-tuning, we also implemented a more systematic hyperparameter optimization strategy using nested cross-validation. Specifically, a 5-fold outer cross-validation was used to evaluate model performance, while an inner 3-fold cross-validation was employed to search over a parameter grid. The hyperparameters explored included learning rate, batch size, number of epochs, and weight decay. For each configuration, the model was fine-tuned and evaluated, and the configuration that maximized the F1-score on the validation sets was selected:

- `learning_rate`: [5e-5, 3e-5, 2e-5]
- `per_device_train_batch_size`: [8, 16]
- `num_train_epochs`: [3, 4]
- `weight_decay`: [0.0, 0.01]

This nested cross-validation process ensured that the hyperparameters were optimized without introducing bias from the test set, while providing a more reliable estimate of generalization performance. We applied this procedure to all Transformer models, in both dataset variants (with identifiers and without identifiers), ensuring consistent and comparable fine-tuning across all experiments.

Once training was complete, each model was saved and later evaluated on the independent test set. Predictions were analysed using confusion matrices and the main evaluation metrics, and misclassified instances were stored for later inspection. This methodology provided a robust and reproducible fine-tuning process that allowed us to benchmark Transformer models against HMM and LSTM approaches in a fair and consistent manner.

4.3.2 Explanation models

In addition to classification, we also train models that provide explanations at the token level. Their goal is not to decide whether a snippet is vulnerable or not, but to annotate each token with a hidden state that helps understand how vulnerabilities manifest in IL. For this purpose, we implemented two models: a supervised HMM and a MEMM.

Supervised HMM

In contrast to the unsupervised HMMs, the supervised HMMs were trained directly from the annotated `.ann` files, where each token is explicitly paired with its corresponding hidden state. This allowed us to estimate the model parameters in a straightforward maximum likelihood fashion, avoiding the need for iterative expectation-maximization.

The training procedure consisted of the following steps:

1. Load annotated files, splitting each token into (*observation*, *state*) pairs.
2. Reverse each line to maintain consistency with the IL representation order.
3. Aggregate all training sequences to compute:
   - **Start probabilities**: frequency of initial states in annotated sequences.
   - **Transition probabilities**: counts of state-to-state transitions.
   - **Emission probabilities**: counts of observation-to-state emissions.
4. Apply add-one smoothing across all distributions to handle unseen events.

The decoding step relies on the Viterbi algorithm. Given a sequence of IL tokens, the algorithm computes the most probable sequence of hidden states, using the supervised transition and emission probabilities as input. This ensures that every prediction aligns with the annotation style used during training.

We trained two separate supervised HMMs: one on annotated vulnerable files and another on annotated non-vulnerable files. This separation reflects the assumption that the class label is already known at the explanation stage, as the classification task is handled by the models presented in Section 4.3.1.

The supervised HMMs offer a generative view of the annotated data, complementing the discriminative MEMMs. They allow us to capture both the probabilistic structure of state transitions and the correspondence between tokens and hidden states. This structure provides an interpretable foundation for the explanation models, since each state sequence can be directly traced back to the annotated ground truth used for training.

MEMM

To train the MEMM we relied on the annotated files (`.ann`), where each token in the IL representation is paired with a hidden state that was manually defined. This format is particularly suitable

for MEMM training, since the model explicitly conditions the probability of the current state not only on the current observation but also on the previous state.

The training pipeline begins with the loading of the annotated files, where tokens are reversed and split into (*observation*, *state*) pairs. From these sequences we extract features of the form:

- `observation`: the current token;
- `previous_state`: the previous hidden state.

These features are then vectorized and used to train a `LogisticRegression` classifier inside a scikit-learn pipeline. The MEMM therefore learns to predict the probability distribution over states given both the observation and the preceding state. To account for imbalanced classes, the logistic regression was trained with `class_weight=balanced`.

As with the HMM models, two MEMMs were trained independently: one using only the annotated vulnerable files, and another using the non-vulnerable ones. This separation reflects the assumption that at the explanation stage we already know which class the snippet belongs to, since the classification task is handled by the best-performing classifiers described earlier.

For inference, we use a Viterbi-style decoding adapted to MEMMs. Given a sequence of observations (tokens), the algorithm computes the most probable sequence of hidden states by recursively combining the probabilities from the logistic regression classifier. This allows us to reconstruct not only the classification decision, but also the underlying token-level path of hidden states that led to it.

This design makes MEMMs a suitable choice for the explanation models, since they are inherently discriminative and state-dependent. By observing how tokens are mapped to hidden states and how transitions are selected, it is possible to provide a detailed justification for why a snippet is considered vulnerable or not. In this sense, MEMMs complement the HMMs: while HMMs capture generative probabilistic structure, MEMMs provide a more flexible discriminative mapping from observations to states, enhancing the explanatory power of our framework.

4.4 VulNLan tool creation

To combine all the previously described components into a single workflow, we developed a tool called **VulNLan**. Its main purpose is to automate the complete pipeline, from source code input to vulnerability classification and explanation.

The process starts by translating PHP source code into IL using our translator. Each snippet is then passed to the classification stage, where three independent models are used in parallel: LSTM, Transformer, and Unsupervised HMM. Their raw outputs (logits or scores) are normalized into probabilities, ensuring comparability across models.

To integrate the predictions, VulNLan applies a heuristic confidence function, which weights each model's probability distribution by its validation accuracy. This results in a combined confidence score for both vulnerable and non-vulnerable classes, expressed both as raw scores and

percentages. The heuristic paired with the decision table is designed to emphasize the most reliable models while still leveraging the complementary strengths of all classifiers, while choosing the final class to be passed onto the explanation models.

With the results from classification step, the supervised HMM and MEMM are loaded and applied at the token level, producing interpretable annotations that highlight how vulnerabilities manifest in IL.

In summary, VulNLan orchestrates the entire workflow: (i) translation, (ii) classification, (iii) heuristic-based decision, and (iv) explanation. This makes it not only a detection tool, but also an interpretability framework to assist developers in understanding vulnerabilities in their code.

4.5 Classification and Interpretation

Classification and interpretation are the two stages that are shown to the user on our tool output. In the classification stage, three models (LSTM, Transformer, and Unsupervised HMM) are trained in parallel to estimate the probability of each snippet being vulnerable or not vulnerable. Their outputs are combined through a heuristic confidence function, which assigns weights based on the accuracy of each model, producing a final decision and showing the full itemized output. However, classification alone is insufficient in a practical vulnerability detection and explanation setting.

For the results to be actionable, developers need to understand why a snippet was classified as vulnerable or not vulnerable. Therefore, in the interpretation stage, supervised models (HMM and MEMM) are employed to analyse code at a finer granularity, annotating tokens and showing the taintedness propagation throughout the code snippet. The output also contains the code snippet annotated with the states. With this output the developer can interpret the results and fix the vulnerability, if one was present on the code.

By classifying the code snippet and showing the states associated with each token, a developer can interpret whether the classification decision was reliable, and find, if present, a vulnerability in the code just by executing the tool and looking at the output on the command-line interface.

4.5.1 Heuristic importance

The heuristic, which in our solution acts as a bridge between the classification and explanation models, allows for a weighted integration of the individual models, combining their strengths while mitigating weaknesses. By coupling confidence scores with majority voting, the system achieves both probabilistic interpretability and robustness to individual model errors. Furthermore, the approach remains flexible, allowing the inclusion of additional models or updated parameters in future iterations, thus ensuring adaptability as more advanced pre-trained models become available. The result of the heuristic, which is calculated through Equations 3.1 and 3.2, combined with the decision table (Table 3.2) outputs the class for any VulNLan user and gives the explanation layer the class in which the model will do their task of labelling token-by-token.

Chapter 5

Evaluation & Results

In this chapter, we evaluate the approach, showcase the results obtained, and discuss the strenghts and limitations of our solution, VulNLan. As explained in Section 4.3.1, we managed to fine-tune the models by interpreting the results with a certain set of hyperparameters. This is where we show the interpretation of such results. The final results are evaluated to check the effectiveness of our approach.

This chapter starts with Section 5.1, where we evaluate the changes done to the dataset throughout our approach. Section 5.2 describes the development environment and our training and testing procedures. Section 5.3 is where we unfold the results of each model, followed by Section 5.4, where we test and evaluate VulNLan. Lastly, 5.5 we evaluate the overall approach and discuss its strengths and limitations.

5.1 Datasets Analysis

In this section, we address if the chosen dataset was appropriate for the tasks developed, namely, classification and explanation. We evaluate the changes done to the dataset throughout the implementation stage for each dataset, respectively, IL and annotated datasets. The section ends with the limitations of our datasets.

As described on Section 4.2, before advancing with the tool implementation, we calculated the similarity of two versions of our IL dataset, one with obfuscated tokens identifiers and the other without those identifiers. For our tool we also created an annotated dataset, labelled token-by-token, which we are going to analyze in this section

5.1.1 Statistical overview

The dataset chosen contains a total of 1,362 PHP snippets, of which 502 are labeled as vulnerable and 860 as non-vulnerable, as shown in Table 4.1. From this original source code dataset, we derived three datasets: two unlabelled IL datasets (one with identifiers preserved and another without identifiers), and one fully annotated IL dataset with token-level labels.

The distinction between keeping or removing identifiers has a direct impact on the statistical properties of the dataset. When identifiers are preserved, the vocabulary becomes larger and more

diverse, which can increase model memorization but reduce generalization. On the other hand, removing identifiers reduces vocabulary size, leading to lower similarity range across snippets, but potentially producing more robust models that focus on structural rather than lexical patterns. As expected, the dataset with identifiers showed a higher similarity range and almost double the standard deviation compared to the obfuscated version (Table 4.2).

Finally, the annotated dataset, although smaller in absolute size, plays a different role. Instead of being used for direct classification, it provides token-by-token supervision, enabling sequence labeling methods such as MEMM and HMM. This makes it complementary rather than directly comparable to the IL datasets, since its purpose is to support explanation-oriented models rather than overall classification.

5.1.2 Vocabulary reduction and preprocessing impact

Throughout the preprocessing pipeline, several transformations were applied to the raw PHP files before producing the IL datasets. These included the removal of comments, static strings, and dataset-specific content such as SQL queries, as well as the obfuscation of identifiers (variables, classes, and functions). Each of these steps contributed to a reduction in vocabulary size, simplifying the dataset and removing tokens that could introduce bias or lead to model overfitting.

The removal of comments and static strings was particularly relevant because such tokens often carry human-readable or dataset-specific information that does not generalize to real-world scenarios. For instance, PHP relevant tokens inside comments or hard-coded strings might provide misleading shortcuts for a classifier, enabling it to memorize patterns that are unrelated to actual vulnerability semantics. Their removal ensured that the models focused only on program structure and control flow.

The obfuscation of tokens, such as variables, functions and classes names, that were mapped to generic placeholders (e.g., `varX`, `functionX`), also contributed to the reduction of vocabulary in our models. This transformation prevented the models from overfitting to specific naming conventions, while still preserving the relative order and structural role of these elements within a snippet. As a result, the models trained on obfuscated data had fewer lexical cues to rely on and were forced to learn more abstract patterns.

Overall, the preprocessing pipeline created a more compact and semantically consistent representation of the code. While some contextual richness was inevitably lost (e.g., descriptive variable names that could hint at data provenance), the vocabulary reduction contributed to lowering noise and improving the robustness of the models. This trade-off between information loss and generalization is a central aspect of preparing datasets for vulnerability detection tasks.

For the explanation task, we decided to use the obfuscated IL dataset without identifiers because we do not want an unnecessary large vocabulary for our sequence models. Then, the dataset was annotated manually where each token, was labelled with one of the four states we defined: `taint`, `ntaint`, `san` and `und`. This states were chosen specifically for this particular dataset. Different datasets with a larger vocabulary could benefit from more states. While annotating this

dataset requires human interpretation, this is favourable for our explanation models to do the task of finding the hidden states on new IL code snippets.

5.1.3 Limitations

The dataset used had limitations, that did not improve our models. The limitations that we encountered are the following:

- **Equal files:** some files in the dataset are equal to each other (i.e., with similarity equal to 1), thus presenting to our models files with the same patterns and do not add any contextual value;
- **Relatively low number of instances:** the dataset is composed of $1,362$ files, which could be improved for better training. With more distinct instances, the models are able to train with a broader set of patterns, improving real-world vulnerability detection.
- **Unbalanced dataset:** the dataset is composed of 860 vulnerable files, and just 502 non-vulnerable ones, making it significantly unbalanced. For our training to be balanced we need to discard at least 368 vulnerable files, therefore not making use of the full dataset.
- **Human interpretation for annotating:** to annotate the dataset manually, the files need to be interpreted by a human. As every human being is prone to make mistakes. Consequently, the models might not be fed with the best annotations.

5.2 Experimental Setup

In this section we describe our experimental setup (in terms of hardware and software), followed by the procedures for training, testing, and fine-tuning our models.

5.2.1 Hardware and software environment

The hardware used to develop VulNLan was a desktop, with Central Processing Unit (CPU), GPU, and Random Access Memory (RAM) specifications on Table 5.1.

Table 5.1: Hardware specifications of the experimental environment.

Component	Specification
CPU	Intel Core i7-11700K (8 cores, 16 threads)
GPU	NVIDIA RTX 3060 Ti (8GB VRAM)
RAM	32 GB DDR4

With our dataset size, this hardware was able to handle all tasks in our tool. The implementation was done using Python (version 3.12.4) and was tested on both Linux and Windows operating systems. It relied on a set of widely used libraries for machine learning, natural language processing, and data handling. For deep learning tasks, such as the LSTM and Transformer

models, we used `PyTorch` together with the HuggingFace `Transformers` library, which provides state-of-the-art architectures and training utilities. Traditional machine learning approaches, including the HMM and MEMM models, were developed with `scikit-learn`, complemented by `hmmlearn` for probabilistic sequence models.

Data preprocessing and manipulation steps made use of `NumPy`, `pandas`, and regular expressions, while visualization relied on `matplotlib`. Additional utilities such as `joblib` were employed for model persistence. Finally, our translation module `PHP2IL` module, which integrates `nltk` for tokenization, was responsible for translating PHP code into our IL representation.

Overall, these dependencies ensured that the system combined deep learning, probabilistic models, and custom preprocessing within a cohesive and efficient framework.

5.2.2 Training and testing procedures

The training and testing procedures of all models were described in detail in Section 4.3. Here, we briefly summarize the experimental procedures to highlight the aspects most relevant to the evaluation of results.

All models were trained and tested under the same controlled conditions, using the balanced datasets described in Section 4.2. Data was split into training, validation, and test sets, ensuring stratification to preserve class balance. For every experiment, seeds were fixed to enhance reproducibility, and models were trained iteratively with early stopping or validation-based monitoring to prevent overfitting. With each cycle of training testing and fine-tuning (as represented on Table 3.2), we examined the validation loss, which, for a proper training, needs to decrease between validation steps. In the case it is higher than the previous validation step, the training is not correct and we should change the training hyperparameters.

For the classification task, three types of models were considered: probabilistic (HMM), recurrent neural networks (LSTM), and Transformer-based architectures. Each model was trained with its corresponding dataset format, as explained in Chapter 4. In all cases, final evaluation was performed on the held-out test set.

For explanation tasks, supervised HMM and MEMM models were trained separately on annotated vulnerable and non-vulnerable files. At evaluation time, the appropriate explanation model was selected according to the classification models heuristic paired with our decision table, ensuring consistency in the two-stage pipeline.

The performance of all models was assessed using standard classification metrics: accuracy, precision, recall, and F1-score. For explanation models, the same metrics were used, but calculated as a macro mean at token-level. This protocol ensures that the differences reported in the following sections stem from the models themselves rather than inconsistencies in the training or evaluation process.

5.3 Results

In this section, we present the results obtained from the experiments conducted with the different models. The goal is to assess their effectiveness in both classification and explanation tasks, comparing the performance of traditional probabilistic approaches with deep learning and Transformer-based methods.

The results are structured as follows. First, we report the outcomes of the classification experiments, highlighting the performance of each model on the vulnerable/non-vulnerable prediction task. We then move to the explanation task, where token-level annotations are evaluated in terms of their ability to provide meaningful and consistent insights about vulnerability locations. Finally, we compare the models against each other, analysing their strengths and weaknesses across the two tasks.

This section provides the quantitative basis for the discussion in Section 5.5, where the results are interpreted in light of the research goals and the limitations of the proposed approach.

5.3.1 Evaluation metrics

To evaluate the performance of the models, we relied on standard classification metrics widely used in machine learning and software vulnerability detection. These include:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.3}$$

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.4}$$

Accuracy measures the overall proportion of correct predictions, while precision evaluates the fraction of snippets predicted as vulnerable that were indeed vulnerable. Recall quantifies the fraction of true vulnerabilities correctly identified, and the F1-score corresponds to the harmonic mean of precision and recall, providing a balanced assessment in scenarios where the dataset is not perfectly balanced.

For the explanation models, which operate at the token level, we also report token-level accuracy and consistency with the annotated dataset. These metrics capture how well the models highlight vulnerable or non-vulnerable regions within the code, complementing the global classification perspective.

5.3.2 Classification models

In this subsection, we present the results obtained with the classification models, namely Transformers, LSTM, and the unsupervised HMM. Each family of models was trained and evaluated under the same procedure described in Section 5.2, ensuring fair comparison.

Transformer models

We experimented four Transformer-based models, each trained in two settings: with identifiers preserved and with identifiers obfuscated. This allows us to analyze the impact of vocabulary reduction on model performance. After the cycle of training, testing and fine-tuning, every Transformers model was trained with the hyperparameters: **4** training epochs, a batch size of **8**, warmup steps set to **100**, a weight decay of **0.001**, a learning rate of 1×10^{-5}, and **800** maximum steps.

Table 5.2: Average performance of Transformer models trained with three random seeds, under two configurations: with identifiers and without identifiers on obfuscated code.

Pre-trained model	ID	Accuracy	Precision	Recall	F1-score
`CodeBERT`	yes	0.926	0.964	0.886	0.923
`CodeBERT`	no	0.926	0.964	0.886	0.923
`GraphCodeBERT`	yes	0.926	0.964	0.886	0.923
`GraphCodeBERT`	no	0.930	0.980	0.880	0.925
`RoBERTa`	yes	0.926	0.964	0.886	0.923
`RoBERTa`	no	0.926	0.964	0.886	0.923
`GPT-2`	yes	0.933	1.000	0.866	0.927
`GPT-2`	no	0.953	1.000	0.906	0.951

To account for initialization variance, every model was trained three times with different random seeds, and the results reported in Table 5.2 correspond to the mean values across these runs.

Overall, the results indicate that the impact of identifier removal is model-dependent. `CodeBERT` and `RoBERTa` show almost no difference between the two configurations, suggesting robustness to vocabulary reduction. `GraphCodeBERT` achieves slightly higher precision when identifiers are removed, while `GPT-2` benefits the most from obfuscation, reaching the best overall accuracy and F1-score in the no-identifier setting. These observations highlight that, although identifier information can sometimes help, models pre-trained specifically on source code representations (e.g., `GraphCodeBERT`) or large-scale language modeling (e.g., `GPT-2`) are able to generalize well without it. The model chosen is solely the choice of the tool user because the tool is programmed to use one model of each type. Thus, only one of the eight trained models is used at a time in the heuristic.

Table 5.3: Results of LSTM models with different optimizers.

Optimizer	Accuracy	Precision	Recall	F1-score
`Adam`	0.950	1.000	0.900	0.947
`Adadelta`	0.500	0.000	0.000	0.000
`RMSprop`[1]	0.950	1.000	0.900	0.947

LSTM models

The LSTM architecture was trained with three different optimizers: `Adam`, `Adadelta`, and `RMSprop`. The final hyperparameters of these models are: an embedding dimension of **64**, a hidden dimension of **64**, **5** layers, an output dimension of **2** (suitable for binary classification logits), a dropout rate of **0.5**, a learning rate of **0.003**, **200** training epochs, and a batch size of **77**.

The LSTM architecture was trained once with each of the three optimizers. As shown in Table 5.3, `Adam` achieved the best balance of stability and performance, which motivated its selection for integration in the tool prototype. `RMSprop` produced comparable results with more training epochs, while `Adadelta` failed to converge, resulting in near-random predictions with metrics close to zero. This behaviour illustrates the sensitivity of recurrent models to the choice of optimizer.

HMM unsupervised

We made experiments with an unsupervised HMM, trained directly on the IL sequences. Unlike the supervised HMM, this one does not leverage token labelling, but was instead focused on learning the patterns of vulnerable and non-vulnerable files, with learning performed at class level.

Table 5.4: Results of the HMM unsupervised model at token level.

Model	Accuracy	Precision	Recall	F1-score
HMM unsupervised	0.70	0.70	0.70	0.70

The results (Table 5.4) illustrate the limitations of this approach compared to the other classification models, but they provide a useful baseline for unsupervised vulnerability detection using a probabilistic approach.

5.3.3 Explanation models

In addition to classification, we evaluated models designed to provide token-level explanations, with the goal of identifying which parts of the code contribute to the vulnerability label. Two sequence models were considered: supervised HMMs and MEMMs. Both were trained on the annotated dataset described in Section 5.1, with separate models for vulnerable and non-vulnerable dataset snippets.

HMM supervised

The supervised HMMs were trained directly with the token-level annotations, learning to reproduce the labelling of vulnerable and non-vulnerable tokens. Performance was assessed separately for each class. As shown in Table 5.5, the supervised HMMs achieved balanced performance across both classes, successfully capturing token-level patterns that distinguish vulnerable from non-vulnerable code.

[1] `RMSProp` converged with 400 trainings epochs, unlike `Adadelta`, that did not converge in any training. The results here are with the same hyperparameters described, except the training epochs.

Table 5.5: Results of the supervised HMM explanation model.

Class	Accuracy	Precision	Recall	F1-score
Vulnerable	0.903	0.956	0.907	0.922
Non-vulnerable	0.897	0.914	0.921	0.917

MEMM

The MEMM models were trained with the same annotated dataset, but rely on a discriminative formulation that conditions state transitions on observed features. As with the HMMs, separate models were trained for vulnerable and non-vulnerable snippets.

Table 5.6: Results of the MEMM explanation model.

Class	Accuracy	Precision	Recall	F1-score
Vulnerable	0.903	0.956	0.907	0.922
Non-vulnerable	0.778	0.873	0.803	0.820

Table 5.6 shows that while the MEMM achieved comparable performance to the HMM in the vulnerable class, it lagged behind for non-vulnerable code, particularly in accuracy and recall. This suggests that although discriminative models can better exploit local contextual information, in this case the generative HMMs provided more balanced results across both classes.

5.3.4 Comparative analysis

Bringing the results together, a clear distinction emerges between deep learning models and probabilistic approaches. Transformer-based architectures outperformed the other models on a seed basis. Even if the mean results show lower performance, there were trainings better and worse. `GPT-2` in the no-identifier setting achieving the highest overall accuracy and F1-score. This highlights the benefits of large-scale pre-training, even when the model was not originally specialized for source code.

The LSTM models showed good performance when trained with `Adam` or `RMSprop`, confirming their capacity to capture sequential dependencies, though their reliance on a suitable optimizer was evident from the failure of `Adadelta` to converge. Compared to Transformers, however, LSTM results were very similar, and the tested instances that missed the classification were very similar between the models. By deep diving into those instances we noticed that there were similar instances, with similarity of 1, but with the opposite label, which means that one of those instances is miss-labelled. This missed label results in worse results, even if the training was in perfect conditions.

The unsupervised HMM established a lower but informative baseline. Its limited performance illustrates the difficulty of learning vulnerability patterns without labeled data, yet it provides insight into the potential of unsupervised learning for scenarios where annotations are scarce.

For the explanation task, supervised HMMs demonstrated balanced performance across vulnerable and non-vulnerable classes, whereas MEMMs, despite their discriminative nature, strug-

```
var1 shell_exec
var1 mysql_real_escape_string var1
var2 sprintf SQL_QUERY var1
var3 mysql_connect
mysql_select_db
echo var2
var4 mysql_query var2
loop var5 mysql_fetch_array var4
  print_r var5
  echo
mysql_close var3
```

Listing 5.1: Non-vulnerable file translated to IL.

gled with non-vulnerable snippets. This contrast suggests that generative models may capture more stable token-level patterns in this setting.

Overall, the results indicate that deep learning methods, particularly Transformers and LSTMs, are best suited for classification, while HMMs remain competitive for explanation tasks. This division reflects the complementary strengths of modern neural and deep learning architectures and traditional probabilistic models, providing valuable insights for the discussion in Section 5.5.

5.4 VulNLan tool evaluation

In this section we are going to evaluate the full use of VulNLan with 50 code snippets from the dataset presented on Section 5.4.2. Firstly, we evaluate the full output of the tool when we execute a new PHP snippet on it, which includes the classification, heuristic, and explanation results. Then we provide a full evaluation of our tool, showcasing the overall classifications with 50 code snippets. Regarding heuristic, we test it while testing and evaluating our tool, showcasing the results of Equations 3.1 and 3.2.

5.4.1 Output example

To showcase the output of our tool, with the classification, heuristic, and explanation results, we decided to execute the tool with two code snippets (one vulnerable and one not vulnerable), regarding SQLi vulnerability. For a better explanation of the output, we split it by the models' results. Firstly, we demonstrate the classification model results, followed by the heuristic, and lastly, the explanation output. Before any model is executed, the PHP code snippet is translated to our IL. The original vulnerable snippet used is the one present in Listing 4.1, and its corresponding translation in IL is already represented on Listing 4.2. The not vulnerable snippet translation is represented on Listing 5.1, respectively, for the models LSTM, Transformers (with and without identifiers) and unsupervised HMM.

```
----- LSTM model output: -----

Analysis (LSTM) of: CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
  Prediction: Vulnerable
  Probabilities: Not Vulnerable: 0.19%, Vulnerable: 99.81%

Analysis (LSTM) of:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
  Prediction: Not Vulnerable
  Probabilities: Not Vulnerable: 99.98%, Vulnerable: 0.02%
-------------------------------------
```

Listing 5.2: LSTM output for both snippets executed on our tool.

Classification output

The name of the vulnerable file starts with `CWE_89__popen__no_sanitizing` and the not vulnerable file name starts with `CWE_89__shell_exec`. The output of the classification models is demonstrated on Listings 5.2, 5.3, and 5.4.

The first results that are shown in the output are obtained from the LSTM model, that we present on Listing 5.2. The results with the two snippets are very close to 100%, and, correctly, classified the first snippet as vulnerable and the second with not vulnerable.

The second result that make our output is the Transformer models output, that we represent on Listing 5.3. In this specific case we decided to join the output for the same snippets using two different Transformers models. The first two results (one for each snippet), are executed on a `CodeBERT` pre-trained Transformers model trained using the dataset with identifiers (IDs) on the code obfuscation tokens, while the last two results use the same pre-trained model but trained over with a dataset that does not show the identifiers. The results between the first Transformers result and the second are similar, only changing 0.02% for code snippet. These competitive results can be achieved with and without identifiers. So even with variables, classes and functions names fully obfuscated, the Transformers architecture can still learn the patterns that compose the file, and recognize if code is vulnerable or not vulnerable.

The last classification model, and coincidently, the one that performed worse, is the unsupervised HMM, that we showcase the respective results in Listing 5.4. While the Transformers models and the LSTM had results very close to 100%, this model, even if correctly classified the snippets, it was not with the best probabilities (i.e., 80.63% vulnerable probability for the vulnerable file and 67.29% not vulnerable probability for the not vulnerable file). This was expected due to a lower accuracy during testing.

Heuristic output

The classification results provided the values for our heuristic formulas (Equations 3.1 and 3.2). The results paired with the models accuracy (accuracy for LSTM model was 0.93 at time of ex-

```
----- Transformer model (with ID) output: -----
Analysis (Transformer) of:
    CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
  Prediction: Vulnerable
  Probabilities: Not Vulnerable: 0.03%, Vulnerable: 99.97%

Analysis (Transformer) of:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
  Prediction: Not Vulnerable
  Probabilities: Not Vulnerable: 99.93%, Vulnerable: 0.07%
-------------------------------------

----- Transformer model (without ID) output: -----
Analysis (Transformer) of:
    CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
  Prediction: Vulnerable
  Probabilities: Not Vulnerable: 0.05%, Vulnerable: 99.95%

Analysis (Transformer) of:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
  Prediction: Not Vulnerable
  Probabilities: Not Vulnerable: 99.95%, Vulnerable: 0.05%
-------------------------------------
```

Listing 5.3: Transformer classification output using VulNLan with Transformers model trained with and without identifiers, respectively.

ecution, even we did better training and achieved 0.95 as represented on Table 5.3, Transformers model was 0.95 and unsupervised HMM was at 0.70) allow us to calculate the heuristic values. The results are shown in the tool output, that were the following in Listing 5.5.

The results of the heuristic in Listing 5.5 show that, with high confidence, the first snippet is vulnerable and the second snippet is not vulnerable. Even if the confidence value of the second snippet is less than 80%, the value for vulnerable is just 7.67%. In this specific case, we notice that every classification model produced the same results for each snippet. So here we do not have to apply the majority voting on Table 3.2.

Explanation output

For the explanation, we use the supervised models at token-level. We present results for both supervised HMM and MEMM models. To check if the explanation was done correctly, we need to manually analyse the taintedness propagation. If the file is not vulnerable we should be able to see the `taint` state transition to `ntaint` when passing a sanitization token, that has the state `san`.

In both Listings 5.6 and 5.7, we demonstrate the outputs of our explanation models. For the first snippet, the `taint` state is propagated through the file, and reaches a sensitive sink, in this case `mysql_query`. For the second file, both models recognize that, in the first line, the input is

```
----- HMM unsupervised model output: -----
Analysis (HMM unsupervised) of:
    CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
  Prediction: Vulnerable
  Probabilities: Not Vulnerable: 19.37%, Vulnerable: 80.63%

Analysis (HMM unsupervised) of:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
  Prediction: Not Vulnerable
  Probabilities: Not Vulnerable: 67.29%, Vulnerable: 32.71%
-----------------------------------
```

Listing 5.4: Unsupervised HMM output for both snippets executed on our tool.

```
Snippet: CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
  Heuristic Confidence (Non-Vulnerable): 0.14 (4.67%)
  Heuristic Confidence (Vulnerable): 2.44 (81.33%)

Snippet:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
  Heuristic Confidence (Non-Vulnerable): 2.35 (78.33%)
  Heuristic Confidence (Vulnerable): 0.23 (7.67%)
```

Listing 5.5: Heuristic results of both snippets tested.

tainted (`taint`), but in the second line, after going through the sanitization function (`san`), the tainted content is sanitized and becomes not tainted (`ntaint`). When it reaches the sensitive sink, with the state `ntaint`, the vulnerability SQLi is not present.

The supervised HMM, in Listing 5.6, we can see a clear propagation of states. In the first snippet, the `taint` state is propagated through the code and, because of the lack of sanitization, it reaches the sensitive sink `mysql_query`. In the second code snippet, the `taint` state is sanitized (line 18) and does not reach the sensitive sink (line 23), i.e., the state changes to `ntaint` and reaches the sink as such.

The MEMM model produced similar results in Listing 5.7, only differentiating on the last line of the second snippet, where it chose the state `und` for both tokens. This difference is irrelevant for vulnerability detection in this snippets, because we have interest between the input and the sensitive sink, where the tainted input can be sanitized to avoid the SQLi vulnerability.

5.4.2 VulNLan testing

To test our tool, we decided to execute it with more test instances. Here we present the testing done with 25 vulnerable and 25 not vulnerable code snippets, to assess the tool efficacy.

Table 5.7 represents the tests done on our tool. Each row represents a class of snippets, and columns show the Correct and Incorrect (Cor. and Inc., respectively) predictions of the differents

```
----- HMM supervised model output: -----
Ficheiro: CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
var1::taint  popen::taint
var2::taint  fread::taint  var1::taint  4096::und
pclose::und  var1::taint
var3::taint  SQL_QUERY::taint  var2::taint
var4::taint  mysql_connect::und
mysql_select_db::und
echo::und  var3::taint
var5::taint  mysql_query::taint  var3::taint
loop::taint  var6::taint  mysql_fetch_array::taint  var5::taint
print_r::taint  var6::taint
echo::und
mysql_close::taint  var4::taint

Ficheiro:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
var1::taint  shell_exec::taint
var1::ntaint  mysql_real_escape_string::san  var1::taint
var2::ntaint  sprintf::ntaint  SQL_QUERY::ntaint  var1::ntaint
var3::und  mysql_connect::und
mysql_select_db::und
echo::und  var2::und
var4::ntaint  mysql_query::ntaint  var2::ntaint
loop::ntaint  var5::ntaint  mysql_fetch_array::ntaint  var4::ntaint
print_r::ntaint  var5::ntaint
echo::und
mysql_close::ntaint  var3::ntaint
-------------------------------------
```

Listing 5.6: Supervised HMM output for both snippets where we can check the taintedness propagation.

models and heuristic that integrate the tool. In general, we achieved positive results, and excluding the unsupervised HMM for non-vulnerable class, the worst model predicted the wrong class only twice. The unsupervised HMM gave 9 wrong predictions for the non-vulnerable class, which is, the least accurate model.

For the vulnerable class, which is composed of 25 vulnerable snippets, every classification model missed 2 classifications. By analysing it forward we checked that for one snippet, every model predicted the wrong class. This specific snippet has in its content a sanitization function, but it is marked as unsafe in the original dataset. A snippet with the same sanitization was the origin of the second misclassified snippet. But, differently to the first wrong snippet, the unsupervised HMM recognized it as vulnerable, and classified other snippet as not vulnerable. The explanation for this class did the transition from the state `taint` to `ntaint` when it passed through the sanitization function. The incorrect results of the explanation models are from the same two code snippets that were also incorrectly classified by the models, and consequently, by the heuristic. In these explanation models, the state that reached the sink is `ntaint` because input is sanitized.

```
----- MEMM supervised model output: -----
Ficheiro: CWE_89__popen__no_sanitizing__select_from_where-
    interpretation_simple_quote.il
var1::taint popen::taint
var2::taint fread::taint var1::taint 4096::und
pclose::und var1::taint
var3::taint SQL_QUERY::taint var2::taint
var4::taint mysql_connect::und
mysql_select_db::und
echo::und var3::taint
var5::taint mysql_query::taint var3::taint
loop::taint var6::taint mysql_fetch_array::taint var5::taint
print_r::taint var6::taint
echo::und
mysql_close::taint var4::taint

Ficheiro:
    CWE_89__shell_exec__func_mysql_real_escape_string__multiple_select-
    sprintf_u_simple_quote.il
var1::taint shell_exec::taint
var1::ntaint mysql_real_escape_string::san var1::taint
var2::ntaint sprintf::ntaint SQL_QUERY::ntaint var1::ntaint
var3::und mysql_connect::und
mysql_select_db::und
echo::und var2::und
var4::ntaint mysql_query::ntaint var2::ntaint
loop::ntaint var5::ntaint mysql_fetch_array::ntaint var4::ntaint
print_r::ntaint var5::ntaint
echo::und
mysql_close::und var3::und
-------------------------------------
```

Listing 5.7: MEMM output for both snippets where we can check the taintedness propagation.

In the not vulnerable class (second line of Table 5.7) we have a larger dispersion of classifications. The Transformers model correctly predicted every snippet, while LSTM and unsupervised HMM missed 2 and 9 classifications, respectively. After analysing every classification, the wrong classifications from LSTM and unsupervised HMM were on different snippets. For each one of the 25 code snippets, they had, at most, one wrong classification, meaning that each code snippet had 2 or 3 correct classifications. Based on the heuristic and decision table, this is the reason of all correct heuristic classifications on the not vulnerable test set. The explanation models also correctly predicted the taintedness propagation of all snippets.

In Table 5.8, we present the confusion matrix for the heuristic and for the explanation models, which had the same results. The two instances missed are the files marked as vulnerable that have sanitization functions in their content.

The metrics that evaluate the confusion matrix in Table 5.8 are represented in Table 5.9. These metrics evaluate the overall performance of our tool, in terms of classification and explanation. The accuracy had a promissing value of 0.960, with a precision of 1.000 because we do not have any false positives. Recall of 0.926 shows that a small number of vulnerable cases were misclassified

Table 5.7: Classification and explanation results of the VulNLan tool.

Class	Tran.		LSTM		HMM uns.		Heuristic		HMM sup.		MEMM	
	Cor.	Inc.	Cor.	Inc.	Cor.	Inc.	Cor.	Inc.	Cor.	Inc.	Cor.	Inc.
Vuln	23	2	23	2	23	2	23	2	23	2	23	2
NVuln	25	0	23	2	16	9	25	0	25	0	25	0

Table 5.8: Confusion matrix of the heuristic and explanation models.

Predicted / True	Vulnerable	Non-vulnerable
Vulnerable	23	0
Non-vulnerable	2	25

as non-vulnerable, which slightly reduced the sensitivity of the model. Nevertheless, the resulting F1-score of 0.962 demonstrates a balanced performance, highlighting that the tool achieves very high precision while maintaining a strong ability to correctly identify vulnerable instances.

5.5 Discussion

In this section, we reflect on the outcomes of our work. We begin by outlining the strengths of the proposed approach, then discuss the main limitations and challenges that arise from our methodology.

5.5.1 Strengths of the approach

The experimental results demonstrate the viability of combining deep learning and probabilistic models for software vulnerability detection. Transformer-based models achieved strong performance in the classification task, with `GPT-2` and `GraphCodeBERT` showing particular robustness even under identifier obfuscation. This highlights the ability of large-scale pre-training to capture semantic patterns that generalize beyond surface-level tokens. The results between the Transformers models with identifiers on the obfuscated code and models without those identifiers are similar. For this architecture, the presence of identifiers does not change the pattern recognition of the model.

At the same time, the explanation task benefited from more traditional approaches. Supervised HMMs were able to provide balanced token-level annotations, capturing both vulnerable and non-vulnerable regions with high accuracy. This indicates that probabilistic models, while less competitive in global classification, remain valuable when fine-grained interpretability is required.

VulNLan showed good and promising results after testing, just missing the classification of two code snippets, that were labelled originally as unsafe, but used a sanitization function to validate the input.

Overall, the combination of neural and probabilistic approaches produced complementary insights: deep learning excels at prediction, while sequence models provide structured explanations

Table 5.9: Results of the heuristic and explanation models.

Accuracy	Precision	Recall	F1-score
0.960	1.000	0.926	0.962

that are easier to interpret by human analysts.

5.5.2 Limitations and threats to validity

Despite these encouraging results, several limitations must be acknowledged:

- **Dataset size and representativeness:** the annotated dataset, while sufficient for experimentation, may not capture the full diversity of real-world vulnerabilities, limiting the external validity of the findings.
- **Incorrect labelling:** when a file is labelled incorrectly, it compromises the results. While building labelled dataset, the labels must be precise.
- **Optimizer and hyperparameters:** the choice of optimizer proved critical for LSTM performance. Other hyperparameters (e.g., learning rate, hidden size) were not exhaustively tuned, potentially underestimating model capacity.
- **Unsupervised baseline:** the unsupervised HMM offered limited performance. While useful as a baseline, it may not represent the full potential of modern unsupervised approaches.
- **Explanation quality:** while token-level accuracy was reported, no human evaluation was conducted to assess whether the explanations align with expert intuition, which may affect practical applicability.

Chapter 6

Conclusion

This dissertation introduced a vulnerability detection and explanation solution. Detecting and classifying vulnerabilities in web application source code, together with providing an explanatory allowed us to build our tool VulNLan. The tool receives web application source code, uses NLP to translate it to IL, and, with classification models combined in the heuristic, provides a explanation with sequence models.

The solution detects, classifies, and further explains SQLi attack code snippets in web application source code that was previously translated to our IL. The input is translated to the IL through our module PHP2IL and classified with our Transformers, LSTM and unsupervised HMM models. The final classification is calculated on the heuristic layer, which provides a weighted by accuracy result, and a decision table for majority voting on the final class. With this decision we execute the HMM and MEMM supervised sequence models corresponding to the final class and output the IL snippet with the states we defined, showing the propagation of taintedness.

To assess the efficacy of our solution we presented the results of every model we trained. The unsupervised HMM model lacked accuracy, with its final value of 0.700. The other classification models performed well, with accuracies greater than 0.900, with several trainings above 0.950. The heuristic benefits the models with better accuracy, therefore, even if unsupervised HMM misclassified a snippet, we have the support of the other models. To assess the explanatory approach, we obtained the results for each class, at token-level. Supervised HMM performed better on the non-vulnerable class, and both supervised HMM and MEMM had the same results in the vulnerable class, with accuracies of 0.903 and F1-score of 0.922. While we tested each model separately, we also tested the full utilization of VulNLan and achieved promising results, more specifically, an accuracy 0.960 and a F1-score of 0.962 on the heuristic and explanation models. We also have to note that the heuristic was able to gather the positive classifications from the models.

We conclude that our solution is able to identify and provide an explanation for the vulnerabilities found in web applications source code. But the true efficacy of the approach only can be evaluated with human evaluation while using the tool.

Our tool can also fit more vulnerability classes if we train the models on datasets containing those type of vulnerabilities. To explain the taintedness propagation we also would have to annotate files that contain these new vulnerabilities, because of the different sensitive sinks.

6.1 Future work

The limitations of our solution point to several avenues for future work. Larger and more diverse datasets, particularly those drawn from industrial codebases, could improve robustness and generalizability. For classification, exploring advanced Transformer variants and fine-tuning strategies may further boost accuracy, such as better pre-trained models trained over web applications source code. For explanation, hybrid approaches that combine neural attention mechanisms with probabilistic sequence models could provide richer and more interpretable insights.

Another promising direction is the integration of semi-supervised, self-supervised or even federated learning, which may overcome the reliance on token-level annotations. Finally, incorporating human interpretation and evaluation of explanations would ensure that the models not only achieve high quantitative scores but also provide insights that are actionable for developers and security analysts.

Bibliography

[1] Wasyihun Sema Admass, Yirga Yayeh Munaye, and Abebe Abeshu Diro. Cyber security: State of the art, challenges and future directions. *Cyber Security and Applications*, 2:100031, 2024. URL: https://www.sciencedirect.com/science/article/pii/S2772918423000188, doi:10.1016/j.csa.2023.100031.

[2] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z. Berkay Celik, Xiangyu Zhang, and Dongyan Xu. ATLAS: A sequence-based learning approach for attack investigation. In *30th USENIX Security Symposium*, pages 3005–3022, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/alsaheel.

[3] Automatic Language Processing Advisory Committee (ALPAC). Languages and machines: computers in translation and linguistics. Technical report, National Academy of Sciences, National Research Council, Washington, D.C., 1966. URL: https://www.mt-archive.net/50/ALPAC-1966.pdf.

[4] Mariette Awad and Rahul Khanna. *Hidden Markov Model*, pages 81–104. Apress, Berkeley, CA, 2015. doi:10.1007/978-1-4302-5990-9_5.

[5] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. doi:10.1109/72.279181.

[6] Elisa Bertino. Data security and privacy: Concepts, approaches, and research directions. In *40th Annual Computer Software and Applications Conference*, volume 1, pages 400–407. ieee, 2016. doi:10.1109/COMPSAC.2016.89.

[7] Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan. Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?, 2023. URL: https://arxiv.org/abs/2306.01754, arXiv:2306.01754.

[8] Chen Dong, Hao Wu, and Qingyuan Li. Multiple Observation HMM-Based CAN Bus Intrusion Detection System for In-Vehicle Network. *IEEE Access*, 11:35639–35648, 2023. doi:10.1109/ACCESS.2023.3265018.

[9] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. doi:10.1207/s15516709cog1402_1.

[10] Ana Fidalgo, Ibéria Medeiros, Paulo Antunes, and Nuno Neves. Towards a deep learning model for vulnerability detection on web application variants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 465–476, 2020. doi:10.1109/ICSTW50294.2020.00083.

[11] Alexandra Figueiredo, Tatjana Lide, David Matos, and Miguel Correia. MERLIN: Multi-Language Web Vulnerability Detection. In *IEEE 19th International Symposium on Network Computing and Applications*, pages 1–9, 2020. doi:10.1109/NCA51143.2020.9306735.

[12] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[13] Jorge Guerreiro and Ibéria Medeiros. Processing web applications using nlp for vulnerability identification. In *2025 20th European Dependable Computing Conference Companion Proceedings (EDCC-C)*, pages 68–71, 2025. doi:10.1109/EDCC-C66476.2025.00033.

[14] Wenbo Guo, Yong Fang, Cheng Huang, Haoran Ou, Chun Lin, and Yongyan Guo. HyVulDect: A hybrid semantic vulnerability mining system based on graph neural network. *Computers & Security*, 121:102823, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0167404822002176, doi:10.1016/j.cose.2022.102823.

[15] Hazim Hanif and Sergio Maffeis. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, page 1–8. IEEE, July 2022. URL: http://dx.doi.org/10.1109/IJCNN55064.2022.9892280, doi:10.1109/ijcnn55064.2022.9892280.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9:1735–80, 12 1997. doi:10.1162/neco.1997.9.8.1735.

[17] W.J. Hutchins. *Machine Translation: Past, Present, Future*. Computers and their applications. Ellis Horwood, 1986. URL: https://books.google.pt/books?id=L1piAAAAMAAJ.

[18] IBM. Cost of a data breach report 2024. https://www.ibm.com/reports/data-breach, 2024.

[19] ISC2. ISC2 Cybersecurity Workforce Study. https://www.isc2.org/-/media/Project/ISC2/Main/Media/documents/research/ISC2_Cybersecurity_Workforce_Study_2023.pdf, 2023.

[20] ISC2. ISC2 Cybersecurity Workforce Study. https://www.isc2.org/Insights/2024/10/ISC2-2024-Cybersecurity-Workforce-Study, 2024.

[21] ITRC. 2023 data breach report. https://www.idtheftcenter.org/publication/2023-data-breach-report/, 2024.

[22] F. Jelinek. *Statistical Methods for Speech Recognition*. Language, Speech, and Communication. MIT Press, 1997. URL: https://mitpress.mit.edu/9780262100663/statistical-methods-for-speech-recognition/.

[23] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 01 1972. doi:10.1108/eb026526.

[24] Karen Sparck Jones. *Natural Language Processing: A Historical Review*, pages 3–16. Springer Netherlands, Dordrecht, 1994. doi:10.1007/978-0-585-35958-8_1.

[25] Arnav Joshi, Ravendar Lal, Tim Finin, and Anupam Joshi. Extracting cybersecurity related linked data from text. In *2013 IEEE Seventh International Conference on Semantic Computing*, pages 252–259, 2013. doi:10.1109/ICSC.2013.50.

[26] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. unpublished, 3rd edition, 2024. Online manuscript released August 20, 2024. URL: https://web.stanford.edu/~jurafsky/slp3/.

[27] Soolin Kim, Jusop Choi, Muhammad Ejaz Ahmed, Surya Nepal, and Hyoungshick Kim. VulDeBERT: A Vulnerability Detection System Using BERT. In *International Symposium on Software Reliability Engineering Workshops*, pages 69–74, 2022. doi:10.1109/ISSREW55968.2022.00042.

[28] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, page 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. URL: https://repository.upenn.edu/handle/20.500.14332/6188.

[29] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1188–1196, Bejing, China, 22–24 Jun 2014. PMLR. URL: https://proceedings.mlr.press/v32/le14.html.

[30] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. *Neural networks: Tricks of the trade*, pages 9–50, 1998. URL: https://doi.org/10.1007/3-540-49430-8_2.

[31] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Data Mining*, page 1–18. Cambridge University Press, 2014. doi:10.1017/9781108684163.

[32] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, July 2022. URL: http://dx.doi.org/10.1109/TDSC.2021.3051525, doi:10.1109/tdsc.2021.3051525.

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *In Proceedings Network and Distributed System Security Symposium*, NDSS 2018. Internet Society, 2018. URL: http://dx.doi.org/10.14722/ndss.2018.23158, doi:10.14722/ndss.2018.23158.

[34] Min Ling and Yiwen Zhang. VulScan: A Vulnerability Detection Model Based on Deep Learning. In *2023 International Conference on Blockchain Technology and Information Security*, pages 88–93, 2023. doi:10.1109/ICBCTIS59921.2023.00021.

[35] Tong Liu, Iza Škrjanec, and Vera Demberg. Temperature-scaling surprisal estimates improve fit to human reading times – but does it do so for the "right reasons"? In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9598–9619, Bangkok, Thailand, August 2024. URL: https://aclanthology.org/2024.acl-long.519, doi:10.18653/v1/2024.acl-long.519.

[36] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. doi:10.1007/s10791-009-9096-x.

[37] Avinash Maurya, Jie Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. Deep optimizer states: Towards scalable training of transformer models using interleaved offloading. In *Proceedings of the 25th International Middleware Conference*, MIDDLEWARE '24, page 404–416. ACM, December 2024. URL: http://dx.doi.org/10.1145/3652892.3700781, doi:10.1145/3652892.3700781.

[38] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2016. doi:10.1109/TR.2015.2457411.

[39] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Statically detecting vulnerabilities by processing programming languages as natural languages. *IEEE Transactions on Reliability*, 71(2):1033–1056, 2022. doi:10.1109/TR.2021.3137314.

[40] Microsoft. Microsoft digital defense report. https://www.microsoft.com/en-us/security/security-insider/microsoft-digital-defense-report-2023, 2023.

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL: https://arxiv.org/abs/1301.3781, arXiv:1301.3781.

[42] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. VulChecker: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium*, pages 6557–6574. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky.

[43] NIST. INFOSEC (Information Security). https://csrc.nist.gov/glossary/term/INFOSEC.

[44] NIST. SAMATE - Software Assurance Metrics and Tool Evaluation. https://samate.nist.gov/, 2024.

[45] Marwan Omar and Stavros Shiaeles. VulDetect: A novel technique for detecting software vulnerabilities using language models. In *IEEE International Conference on Cyber Security and Resilience*, pages 105–110, 2023. doi:10.1109/CSR57506.2023.10224924.

[46] OWASP. OWASP Top 10. https://owasp.org/Top10/, 2021.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[48] Rishi Rabheru, Hazim Hanif, and Sergio Maffeis. A Hybrid Graph Neural Network Approach for Detecting PHP Vulnerabilities. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, page 1–9. IEEE, June 2022. URL: http://dx.doi.org/10.1109/DSC54232.2022.9888816, doi:10.1109/dsc54232.2022.9888816.

[49] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. doi:10.1109/5.18626.

[50] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. URL: https://doi.org/10.1037/h0042519.

[51] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. doi:10.1038/323533a0.

[52] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, 2020. URL: https://arxiv.org/abs/1910.01108, arXiv:1910.01108.

[53] Scikit-learn. Nested versus non-nested cross-validation. https://scikit-learn.org/stable/auto_examples/model_selection/plot_nested_cross_validation_iris.html.

[54] Supriya Shende. Long Short-Term Memory (LSTM) Deep Learning Method for Intrusion Detection in Network Security. *International Journal of Engineering Research and*, V9, 07 2020. doi:10.17577/IJERTV9IS061016.

[55] Hamed Taherdoost. A review on risk management in information systems: Risk policy, control and fraud detection. *Electronics*, 10(24), 2021. URL: https://www.mdpi.com/2079-9292/10/24/3065, doi:10.3390/electronics10243065.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL: http://arxiv.org/abs/1706.03762.

[57] W3Techs. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php, 2024.

[58] W. A. Woods. Transition network grammars for natural language analysis. *Commun. ACM*, 13(10):591–606, October 1970. doi:10.1145/355598.362773.