UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Detecting Web Vulnerabilities in an Intermediate Language by Resorting to Machine Learning Techniques

Documento Provisório

Ana Maria Dias Fidalgo

MESTRADO EM CIÊNCIA DE DADOS

Dissertação orientada por:
Prof. Doutor Ibéria Vitória de Sousa Medeiros
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

2020

Acknowledgments

This work would not be possible without the support and insights from the people around me.

For starters, I want to thank my advisor Prof. Ibéria Medeiros and my co-advisor Prof. Nuno Neves. I could not be happier with the opportunity to learn with them throughout the thesis. They were always eager to learn things outside of their expertise, hear my insights and share theirs. Their enthusiasm motivated and challenged me to achieve higher. To all that, I am incredibly grateful.

Next, I would like to thank my best friends that, even after a lot of crankiness, crying, and complaining throughout this year, they still stick around: Filipa, for always showing me the bright side; Rita, for understanding me better than myself, Sofia, the sister I wish I had; Patrícia, for showing me my value; and André, for never letting me give up and teaching me so much about critical thinking in data science. More than friends; they are my beloved (Lisbon) family.

For the last two years, I also had the pleasure to work alongside excellent people, with whom I shared many lunches, laughs, anxieties, achievements, and the best political arguments: João Lobo, Miguel Silva, Paulo Santos, Guilherme Espada, Francisco Medeiros, and Rita Belo. A special thanks to my mate and friend Pedro Gaspar, one of the kindest and selfless people I know, who is always ready to help me overcome whatever comes in the way.

Life would have certainly been more stressful and sad without Lindy Hop, so a big thank you to the Little Big Apple community for the dances, jams, and good vibes.

And last but not least, thank you to my family for their support and belief: my grandma Isaura and grandpa Armando, for the heartfull family moments they provide every time I go to Coimbra; my mom Céu, who listens to me so patiently; my aunt Palmira, who helped me to continue college when I could not afford it; and my dad, who taught me the value of education and hard work. Even though he is not here, my desire to make him proud still lingers and fuels me.

This work was partially supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference to PTDC/CCI-INF/29058/2017, project SEAL, and LASIGE Research Unit (UIDB/00408/2020), and through P2020 with reference to LISBOA-01-0247-FEDER-039238, XIVT project, an ITEA3 European project (I3C4-17039).

I dedicate this work to my dad, who worked incredibly hard to allow me to thrive.

Resumo

Nos últimos anos, as aplicações web mudaram a forma como os utilizadores usam a Internet e para muitos tornaram-se imprescindíveis para as mais variadas tarefas, desde a socialização a transações bancárias. Com o aumento da sua popularidade, as aplicações web tornaram-se também alvos aliciantes para atacantes cibernéticos, levando a um aumento exponencial do número de vulnerabilidades reportadas. Injeção de SQL (iSQL) é uma vulnerabilidade que ocorre quando o atacante consegue introduzir código SQL num comando que, ao ser executado, produz um efeito diferente do pretendido pelo programador (por exemplo, acesso a dados protegidos ou alteração inadvertida de informação guardada na base de dados). Devido ao seu enorme impacto e facilidade de exploração (basta, por exemplo, introduzir num formulário dados de forma inteligente em conjunto com código SQL), iSQL é das vulnerabilidades mais atrativas e populares. No nosso trabalho decidimos focar-nos exclusivamente neste tipo de fragilidade da web.

Apenas recentemente é que se começou a investigar a utilização de técnicas de Aprendizagem Automática (AA) na deteção de vulnerabilidades (com foco em linguagens como C/C++), e os resultados, apesar de embrionários, são encorajadores, o que nos leva a crer que a aplicação de técnicas de AA nesta área tem um grande potencial. Contudo, para PHP (a linguagem para aplicações web mais popular nos dias de hoje) o trabalho realizado com técnicas de AA é praticamente inexistente. As vulnerabilidades web que ocorrem em linguagens para back-end como PHP tendem a ter um comportamento e características diferentes que linguagens como C/C++, pelo que não se pode concluir que modelos que evidenciaram algum sucesso no passado podem de uma maneira imediata ser aplicados com resultados semelhantes no nosso contexto.

Na nossa abordagem, escolhemos analisar o código PHP numa linguagem intermédia (LI) semelhante ao *Assembly* (para C). Esta opção pretende resolver um dos problemas da deteção de vulnerabilidades em código fonte, relacionada com a quantidade de ruído presente. O código fonte, por ser de alto nível, contém informação sintática necessária à compreensão humana que em nada contribui para o processamento automático e para a tarefa de deteção de vulnerabilidades. Além disso, o uso de uma LI aumenta a flexibilidade da solução, uma vez que pode ser usada para qualquer outra linguagem que possa ser convertida na mesma LI. Mais ainda, uma vez que a LI é uma linguagem de baixo nível quando comparada com a

linguagem do código fonte, é possível ter acesso à estrutura interna das instruções, o que permite a criação e o uso de novas variáveis nos modelos de AA, melhorando o seu desempenho.

A nossa solução passa pela construção de seis conjuntos de dados: o *PHP Excerpt Dataset* (PED), o *Bytecode Excerpt Dataset* (BED), o *Opcode Dataset* (OD), o *Opcode+Operand Dataset* (OOD), o *Slice Dataset* (SD) e o *Simplified Slice Dataset* (SSD). O primeiro contém excertos com e sem vulnerabilidades iSQL, extraídos do repositório público *Software Assurance Reference Dataset (SARD)*. O segundo é constituído pelos excertos correspondentes na LI. O BED foi posteriormente usado para criar os restantes conjuntos de dados. O OD e o OOD são, como o nome indica, conjuntos de dados em que cada exemplo contém apenas os opcodes ou opcodes e operandos do excerto em bytecode, respetivamente. Os últimos dois conjuntos de dados seguem uma abordagem diferente. Para obter uma representação mais próxima da linguagem natural (em que o fluxo de controlo é linear) criámos fatias para cada excerto. Uma *fatia* de um excerto corresponde a um dos seus caminhos de execução, pelo que um excerto pode originar múltiplas fatias.

Para classificar os exemplos, criámos uma rede sequencial de Aprendizagem Profunda (AP). Esta rede é constituída por:

- uma camada de *Embedding*, que recebe uma sequência de vetores e os transforma em vetores capazes de representar a sua informação semântica.

- *n* blocos de camadas *LSTM* e *Dropout*. A camada *LSTM* recebe vetores com informação da camada anterior, codifica padrões relacionados com a ordem dos vetores na sequência, e envia o vetor resultante à camada *Dropout*. A camada *Dropout* escolhe aleatoriamente alguns nós para 'desligar', i.e., modificar os seus pesos para tomarem o valor 0, com o objectivo de prevenir o sobreajuste aos dados. Ao 'desligar' certos conjuntos de nós, a camada *Dropout* previne a adequação do modelo a certos padrões de dados, que podem ter sido aprendidos coincidentalmente, e que têm um impacto negativo no desempenho do modelo, uma vez que não são padrões gerais mas sim padrões específicos de certo subconjunto dos dados de treino.

- uma camada *Dense*, que codifica a relação entre o vetor que recebe e o seu rótulo (vulnerável ou não-vulnerável a iSQL).

- uma camada *Dense* que produz o resultado final. O resultado do modelo é um número real entre 0 e 1, que representa a probabilidade de um exemplo ser vulnerável a iSQL.

Para aferir a perfomance do nosso modelo e a qualidade das representações de cada conjunto de dados, conduzimos uma série de experiências seguindo técnicas de

Ciência de Dados. Começámos por separar cada conjunto em dois subconjuntos, treino e teste. Aplicámos *10-Fold Cross-Validation* ao conjunto de treino 3 vezes, obtendo 30 valores de *accuracy*, *precision* e *reacall* para cada configuração do modelo em cada conjunto de treino. Testámos também diversas configurações do modelo para múltiplos valores dos principais parâmetros: tamanho da camada escondida, número de épocas, taxa de dropout e taxa de aprendizagem. Como este trabalho foi desenvolvido iterativamente e o primeiro conjunto de dados construído e testado foi o OD, a avaliação para este conjunto de dados foi ligeiramente diferente. Primeiro, optámos por usar o valor por defeito da taxa de aprendizagem. Testámos três algoritmos de otimização usados em redes neuronais: ADADELTA, RMSProp e ADAM. Para os restantes conjuntos de dados apenas usámos o RMSProp, uma vez que foi este algoritmo que oblteve melhor desempenho. Para o OD testámos ainda o modelo com 1 e 2 blocos LSTM+Dropout. Concluímos que o ganho no desempenho não era suficiente para justificar o uso de 2 blocos, pelo que decidimos restringir-nos a 1 bloco com os restantes conjuntos.

Contrariamente ao esperado, o OD foi o que obteve melhores resultados (com valores médios acima de 90% em todas as métricas), apesar de conter menos informação do bytecode e não ter controlo de fluxo linear. Ainda assim, os resultados de todos os conjuntos foram bastante bons, sendo que o conjunto com pior perfomance (o OODD) obteve valores médios acima de 60% em todas as métricas.

Uma das conclusões a que chegámos é que seria preciso um conjunto de excertos PHP que espelhasse melhor o paradigma das aplicações web. Os exemplos do PED são muito pequenos e pouco variados, o que nos impede de assegurar a fiabilidade dos resultados obtidos no contexto de aplicações reais. Ainda assim, usámos este conjunto de dados porque não conhecemos bibliografia que tenha um conjunto mais variado e com melhor qualidade. Outro ponto importante que antevemos que melhore os resultados da abordagem com fatias seria a automatização do processo de conversão de excerto para fatias. Infelizmente, por limitações de tempo, optámos por fazer a conversão manualmente de apenas uma parte do BED (aproximadamente um terço). Ainda assim, podemos concluir que esta abordagem tem grandes vantagens uma vez que o desempenho obtido tanto com o SD como com o SSD foi superior ao do OOD.

O planeamento original do projeto foi adaptado no decorrer do mesmo. Inicialmente propusemo-nos estudar diversas arquiteturas de AP, com recurso a diferentes tipos de camadas usadas em Processamento de Linguagem Natural - camadas LSTM, *Convolutional Neural Networks* (CNN) e *Transformers* - mas após obtermos os primeiros resultados, concluímos que era mais relevante fazer um estudo acerca do tipo de informação, e da sua qualidade, que o modelo recebe. Assim, decidimos mudar o foco do nosso trabalho: em vez de fazer um estudo aprofundado sobre o impacto

de diferentes arquiteturas de redes neuronais na qualidade dos resultados, decidimos dar maior ênfase ao estudo do impacto que diferentes representações de informação podem ter nos resultados gerados por diferentes modelos.

Palavras-chave: vulnerabilidades web, deteção de vulnerabilidades, segurança de software, processamento de linguagem natural, aprendizagem profunda

Abstract

The number of vulnerabilities has grown exponentially over the last years, with SQL Injection being especially troublesome for web applications. In parallel, novel research has shown the potential of Machine Learning to find vulnerabilities, which can aid experts to reduce the search space or even classify programs on its own. Previous work, however, rarely includes SQL Injection or considers popular server-side languages for web application development like PHP.

In our work, we construct a Deep Learning model capable of classifying PHP excerpts as vulnerable (or not) to SQL Injection. We use an intermediate language to represent the excerpts and interpret them as text, resorting to well-studied Natural Language Processing techniques. This work can help back-end programmers discover SQL Injection in an early stage of the project, avoiding attacks that would eventually cost a lot to repair their damage.

We also investigate which information should be fed to the model. Hence, we built four datasets (the Opcode Dataset, the Opcode+Operand Dataset, the Slice Dataset, and the Simplified Slice Dataset) from the bytecode dataset that represent each PHP excerpt differently. This approach is a simpler alternative to complex data structures previously used to represent code's control flow. For each of those datasets, we performed several experiments to evaluate alternative configurations for the model. For all datasets, we managed to find a setting that leads to a score, on average, above 60% for the accuracy, precision, and recall.

Keywords: web vulnerabilities, vulnerability detection, software security, natural language processing, deep learning

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The present document constitutes the final work for the Master's in Data Science in the Faculty of Sciences of the University of Lisbon. The work intersects two big fields: Data Science and Software Security. The problem domain - detection of web vulnerabilities in an intermediate language (IL) - belongs to the latter, while the techniques used to collect, clean and model the data, as well as the model evaluation belong to the Data Science field.

In the following sections, we explain why it is crucial to tackling the problem of detecting web vulnerabilities, the advantages of analyzing the code of web applications in an IL, and why we decided to use Machine Learning (ML) techniques to address this issue. Next, we define the goals of this research and state its contributions. To finish, we reveal the structure of the document and give some insights into each chapter.

1.1 Motivation

Web applications have become central in everyone's lives. We use them to check the email, to make transactions, to socialize, and much more. As their role grew, so did their appeal to hackers. That is why the number of web vulnerabilities has continuously grown year by year. SQL Injections (SQLi) are considered to be one of the most devastating web vulnerabilities, as they allow intruders to access and manipulate private data. Also, successful attacks can cost companies much money in repairs. Furthermore, SQLi is relatively easy to exploit, making it even more appealing to attackers.

Although there is some work on vulnerability detection leveraging ML, this area is still at the start and focuses mainly on C/C++. Nevertheless, some research already shows the benefits of using Deep Learning (DL) and Natural Language Processing (NLP) to detect vulnerabilities in the source code [22, 32, 42]. However, there is no previous work experimenting DL in PHP, even though PHP is the most popular server-side language for web applications [51]. Since the vulnerabilities that commonly arise are different for C/C++ and PHP, it is not trivial to assume they would work equally well. Nonetheless, it is something worth considering.

Perhaps the biggest potential with our approach is that it helps to overcome the problem of dealing with source code. Typically, source code has a lot of syntax information useless for the detection task. This problem may be atoned if the analysis is done in an intermediate language (IL). PHP has an IL similar to Assembly (for C/C++), based in opcodes and their operands, which allows us to look closer to the internal structure of the language, which we believe can help in the detection task performed by ML.

1.2 Goals

The main goal of this thesis is to leverage ML to detect web vulnerabilities in an IL. We focused the work on SQLi since the spectrum of web vulnerabilities is vast, and this is the most prominent one. Besides, we decided to analyze PHP excerpts due to the lack of previous research and the relevance it still has. Thus, the main goal was refined into detecting SQLi in an intermediate language for PHP programs, leveraging DL and NLP techniques.

Initially, we thought of building a dataset and experimenting with different DL architectures to determine the best for our dataset. Nonetheless, we understood that the IL had interesting paths we could explore. Consequently, after the first experiments, we built other datasets with different information.

In total, we built six datasets: the PHP Excerpt Dataset (PED), the Bytecode Excerpt Dataset (BED), the Opcode Dataset (OD), the Opcode + Operand Dataset (OOD), the Slice Dataset (SD), and the Simplified Slice Dataset (SSD). The PED was retrieved from the Software Assurance Reference Database (SARD)[1], which provides PHP test cases of both vulnerable and non-vulnerable to SQLi. Each test case is composed of a code excerpt, i.e., an instance of PED. A code excerpt starts in an entry point, an instruction that receives user-defined input, such as *$_GET*. It finishes in a sensitive sink, such as *mysql_query* that, when executed with malicious input, may cause undesired behavior, such as giving access to private data to an unauthorized person. The instructions between the entry point and the sensitive sink can manipulate (or not) the entry point. Each BED sample contains the bytecode for the corresponding PHP excerpt. This dataset is the origin of the following four datasets: OD, OOD, SD, and SSD. These contain a selection of features from the BED and are the ones we used to train the model.

Even though the model itself was not the focus of the thesis, we wanted to build

[1] https://samate.nist.gov/SARD/index.php

a DL network capable of accurately classifying the dataset samples. The model must be capable of extracting relevant features associated with SQLi and the order of the tokens in an excerpt. Hence, our model starts by representing each sample as an embedding vector. Embedding vectors are useful in DL due to their ability to embed semantic information and adapt the dimensionality. Then, the model uses an LSTM layer to extract features related to the order of the elements in the sample. These features are fed to the Dropout layer, which deals with overfitting by zeroing some nodes. Next, the result goes into a Dense layer responsible for learning the relationship between the sample and its label. Finally, the last Dense layer outputs the final value, between 0 and 1, which indicates the probability of the sample being vulnerable to SQLi.

We conducted experiments on the OD, OOD, SD, and SSD. To each, we trained the model with different configurations on 70% of the dataset by performing 10-fold stratified cross-validation repeated three times. Afterwards, we tested the best configuration on the remaining part. In each training, we registered the accuracy, precision and recall. These metrics helped us analyze the model's performance under each configuration, and decide how to proceed with the investigation. All datasets led to models with good performance, in which all the metrics scored, on average, more than 60%. Contrary to our expectations, the OD (the most straightforward dataset) obtained the best results (scores above 90%), followed by the SD, then the SSD, and finally the OOD. Since the performance under the SD and SSD was better than under the OOD, we can conclude that the slice representation we used for them helps the model learning the necessary patterns to SQLi discovery. Nevertheless, we consider that the initial dataset has a significant impact on the results, and that if we manage to build a better one (with more diversified and longer samples, retrieved from real web applications) the scores would be higher on the other data representations.

1.3 Contributions

There are four main contributions of our work:

- The analysis of PHP web applications in the intermediate PHP language;
- A DL network that accurately classifies PHP excerpts as SQLi vulnerable or non-vulnerable;
- Six datasets with different representations for the code excerpts - PED, BED, OD, OOD, SD, and SSD;
- Experimental evaluations providing assessments of different hyperparameter configurations for the datasets OD, OOD, SD, and SSD;

This research led to the publication of the paper *Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants* in the Workshop on Testing of Configurable and Multi-variant Systems co-located with the 2020 IEEE International Conference on Software Testing [18].

1.4 Structure of the document

This document is organized in six chapters, the first one being the introductory chapter we are currently in. The remaining chapters are:

- Chapter 2 - Background and Related Work
  In this chapter, we address two points: which concepts the reader needs to understand the work developed, and what has already been done in previous research that is similar or related to the problem we want to solve and the Data Science challenges that arise from it.

- Chapter 3 - Datasets and Deep Learning Architectures for SQLi Detection
  We start by defining the problem we want to solve. In here, we show how the dataset and model influence the results of the solution. After that, we develop on the approaches followed to create the datasets and the model.

- Chapter 4 - Implementation
  In this chapter, we provide details on how some aspects of our approach were implemented. It is divided into two parts: the first focuses on the datasets creation while the second addresses the network construction.

- Chapter 5 - Experiments
  The experiments are also divided into two parts. In the first part, we show the preliminary results with the Opcode Dataset, where we experimented with different optimization algorithms and one and two LSTM layers. For the second part, we used the results from the first part to choose a single algorithm and work with one LSTM layer.

- Chapter 6 - Conclusion
  To conclude, we discuss the strengths and weaknesses of the present work and propose further research to be taken from here.

Chapter 2

Background and Related Work

Chapter 2 introduces the necessary background and relevant related work on the topics carried by the thesis. The chapter is divided into five subsections: 1) Web Vulnerabilities, 2) Intermediate Language, 3) Deep Learning for Natural Language Processing, 4) Machine Learning in Vulnerability Detection, and 5) Program Representation. The first three sections briefly introduce the necessary theory to understand the solution presented in the next chapter. In the last two, we describe the state of the art of machine learning (ML) usage in vulnerability detection, and of program representation. The state of the art helps us to understand what has already been done, what are the frailties of past work, and what is still left to do.

2.1 Web Vulnerabilities

Vulnerabilities are flaws present in a system. When an attacker exploits them, he can breach some security policy, and the impact can cost a significant amount of money to the organization (e.g., time to fix the damage that was inflicted in the servers). Over the last years, with the increasing importance of the Internet and web applications being widely used, the number of vulnerabilities has grown exponentially [14]. According to the OWASP Top 10 of 2017 [52], the most popular web vulnerability classes are:

1. Injection,
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration

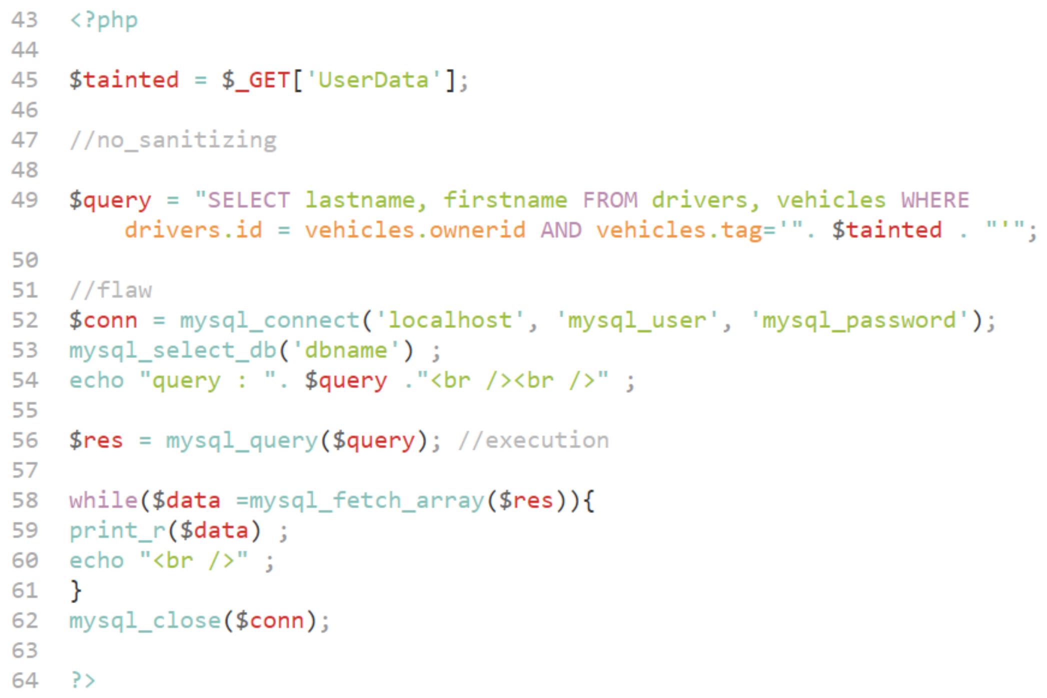
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging&Monitoring

The number one web vulnerability class is called Injection. Injection happens whenever malicious data is sent to a web application, and then an interpreter processes it as part of a command or query (e.g., SQL query). This allows the attacker to trick the interpreter into executing unintended commands or accessing data without proper authorization. An interpreter can be, for instance, accessed by a function of the programming language (e.g., `mysqli_query` on PHP) that receives the injected input as an argument to be included in a query. Injections are easy to exploit because the attacker only has to insert appropriate strings to exploit the target interpreter, usually through the addition of meta characters. The impact of such an attack on the system can be quite high, making these vulnerabilities particularly appealing to attackers. So, the best way to prevent injection vulnerabilities is by guaranteeing that commands and queries are not tainted (compromised) by malicious data. This approach can be made, preferably, by utilizing secure APIs or, if not possible, escaping special characters [14].

There are several types of injections, such as command line, SQL, LDAP, and XML. In this work, we will look at SQL Injection (SQLi) only. According to Clarke [13], SQLi is one of the most devastating bugs. Anytime an application gives an attacker the chance to control SQL queries that it passes to a database, the software is vulnerable to a SQLi vulnerability. This problem is not restricted to web applications, meaning that any system that uses dynamic SQL statements to communicate with a database, like some server-client systems, can be prone to this sort of flaw.

In our work, we chose to detect SQLi vulnerabilities in PHP code, since PHP is the server-side high-level language in which the majority of web applications are written. According to W3Tech [51], 79% of web applications are written in PHP. The PHP example depicted in Figure 2.1 is a SARD vulnerable sample. On line 45, exterior data enters the program through the global array `$_GET`, and it is stored in the `$tainted` variable. On line 49, a query string is constructed with the given input and stored on variable `$query`. Since the input is not evaluated in any way, an attacker could, for example, give as input a string (following the SQL syntax) that, together with the query string, allows the attacker access to private data. The access is obtained through the variable `$res` (line 56), which stores the result from the execution of the query in the database. Therefore, the code has a SQLi vulnerability. Finally, from lines 58 to 61, the value of `res` is displayed to the user.

This vulnerability can be fixed by escaping special characters. For this purpose, there are *sanitization functions* which escape and invalidate metacharacters (e.g., ' and ") that change the structure and the goal of the query. `mysql_real_escape_string` is one of these functions. It can be used in the example to sanitize the `$tainted` variable, before constructing the query. So, line 47 could be replaced by `$tainted = mysql_real_escape_string($tainted)`.



```
43  <?php
44
45  $tainted = $_GET['UserData'];
46
47  //no_sanitizing
48
49  $query = "SELECT lastname, firstname FROM drivers, vehicles WHERE
        drivers.id = vehicles.ownerid AND vehicles.tag='". $tainted . "'";
50
51  //flaw
52  $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
53  mysql_select_db('dbname') ;
54  echo "query : ". $query ."<br /><br />" ;
55
56  $res = mysql_query($query); //execution
57
58  while($data =mysql_fetch_array($res)){
59  print_r($data) ;
60  echo "<br />" ;
61  }
62  mysql_close($conn);
63
64  ?>
```

Figure 2.1: Example of a SARD sample vulnerable to SQLi.

2.2 Intermediate Language

Compilers are software responsible for translating the source code of a program into a low-level language that the machine can execute. ILs were introduced by compiler designers to simplify the translation process. An IL is a data structure representing a program in a simplified manner without losing information. It allows the compiler to break up the program in multiple modules that can be efficiently and independently processed. Moreover, all high-level programming languages that may be represented by the same IL can be dealt equally.

Traditionally, PHP uses a virtual machine engine called Zend [1] to interpret and run PHP programs. Zend transforms the programs into bytecode, which is then interpreted and executed. A tool called Vulcan Logic Dumper (VLD)[2] can intercept

[1] https://www.zend.com/products/php-development-tools

[2] https://github.com/derickr/vld

the bytecode processing before its execution, allowing it to be saved into a file. This way, we gain access to an IL in which the original code *instructions* have been transformed into simpler ones (that we denominate by *statements*) with a more restricted space. Figure 2.2 depicts the bytecode file's structure. At the top of the file, there is the result of the branch analysis. Next, the file presents the bytecode for the main part of the program - the *main code* -, i.e., the code without user-defined classes and functions. The bytecode for the user-defined classes and functions is written below, followed by the branch and path lists.

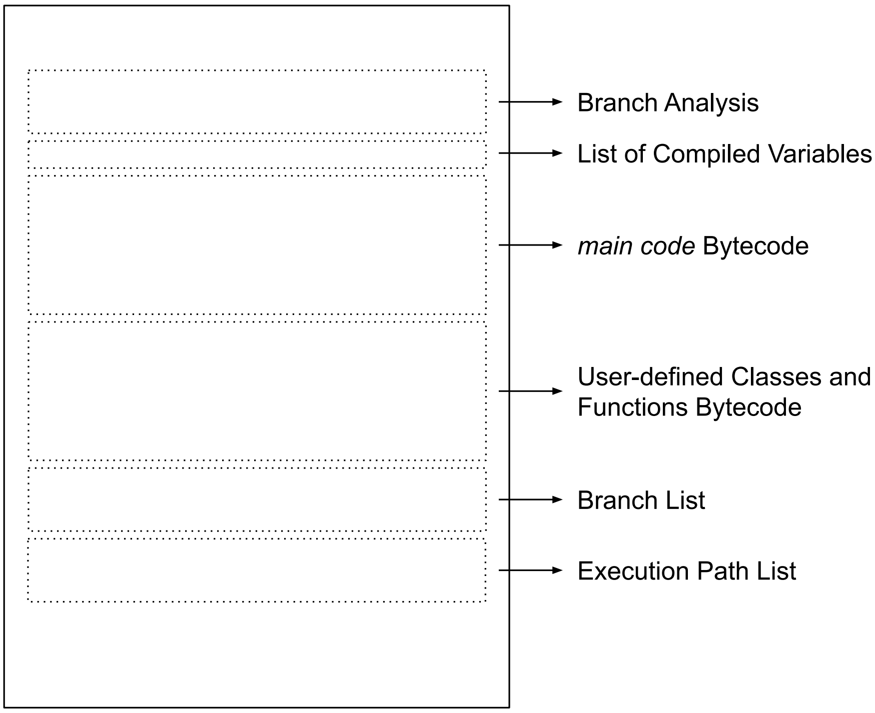
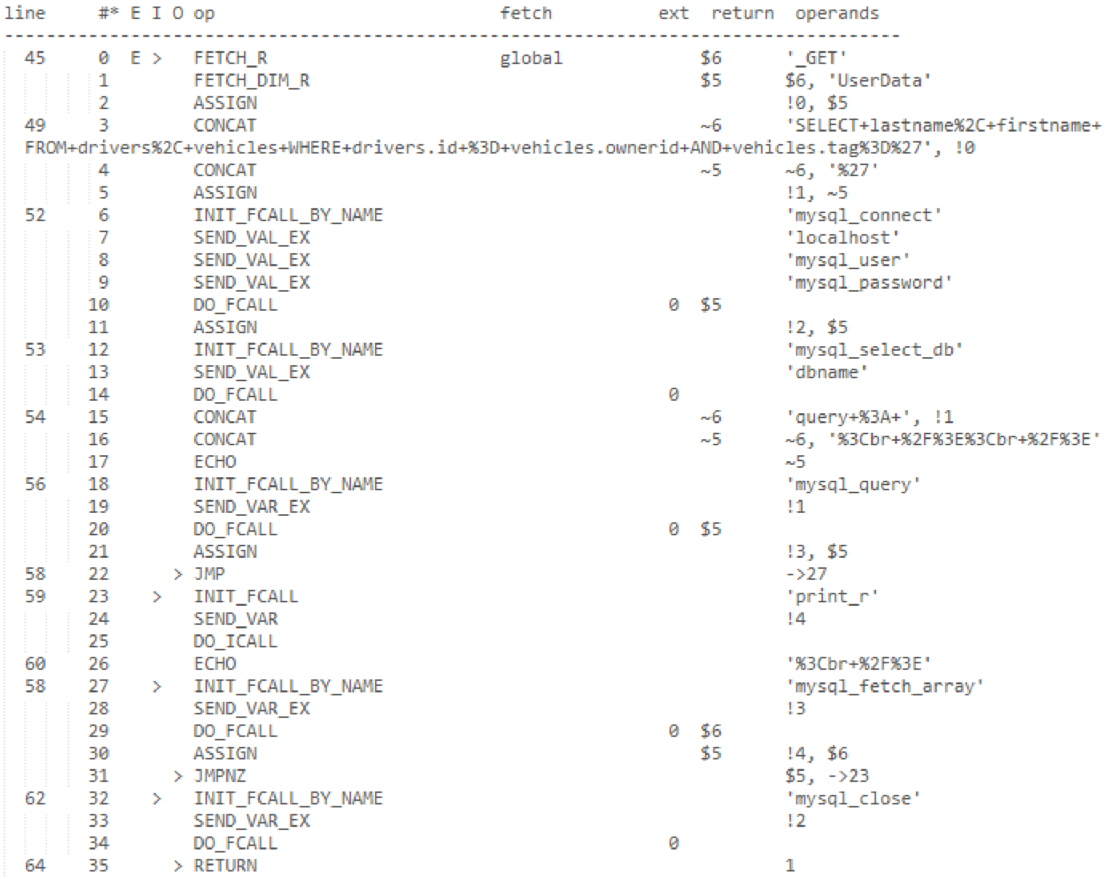

Branch Analysis
List of Compiled Variables
main code Bytecode
User-defined Classes and Functions Bytecode
Branch List
Execution Path List

Figure 2.2: Intermediate language for the example depicted in Figure 2.1.

There are several ILs with different specifications and data structures. Usually, the most used ones are structured (graph or tree-based), tuple-based, stack-based, or combinations of the three. The PHP bytecode is a tuple-based language.

Figure 2.3 shows the main code obtained with VLD for the source code in Figure 2.1. It is easy to understand how this representation is a tuple-based language by considering that each tuple is a line of the table in the figure, in the form `<line,#*, E, I, O, op, fetch, ext, return, operands>`.

Note that, from the bytecode, we can get a lot of insights on how the code is internally executed, which cannot be obtained from the source code. For instance, it shows which are the statements that are executed for each PHP instruction. The first column corresponds to the PHP instruction line and the *op* column to the statement's opcode. An opcode is an elementary operation in the bytecode language. It can have zero, one, or two operands, depicted in column *operands*, where they



```
line     #* E I O op                           fetch          ext  return  operands
-------------------------------------------------------------------------------------
  45     0  E >   FETCH_R                      global                $6      '_GET'
         1        FETCH_DIM_R                                        $5      $6, 'UserData'
         2        ASSIGN                                                     !0, $5
  49     3        CONCAT                                             ~6      'SELECT+lastname%2C+firstname+
  FROM+drivers%2C+vehicles+WHERE+drivers.id+%3D+vehicles.ownerid+AND+vehicles.tag%3D%27', !0
         4        CONCAT                                             ~5      ~6, '%27'
         5        ASSIGN                                                     !1, ~5
  52     6        INIT_FCALL_BY_NAME                                         'mysql_connect'
         7        SEND_VAL_EX                                                'localhost'
         8        SEND_VAL_EX                                                'mysql_user'
         9        SEND_VAL_EX                                                'mysql_password'
        10        DO_FCALL                                        0  $5
        11        ASSIGN                                                     !2, $5
  53    12        INIT_FCALL_BY_NAME                                         'mysql_select_db'
        13        SEND_VAL_EX                                                'dbname'
        14        DO_FCALL                                        0
  54    15        CONCAT                                             ~6      'query+%3A+', !1
        16        CONCAT                                             ~5      ~6, '%3Cbr+%2F%3E%3Cbr+%2F%3E'
        17        ECHO                                                       ~5
  56    18        INIT_FCALL_BY_NAME                                         'mysql_query'
        19        SEND_VAR_EX                                                !1
        20        DO_FCALL                                        0  $5
        21        ASSIGN                                                     !3, $5
  58    22      > JMP                                                        ->27
  59    23    >   INIT_FCALL                                                 'print_r'
        24        SEND_VAR                                                   !4
        25        DO_ICALL
  60    26        ECHO                                                       '%3Cbr+%2F%3E'
  58    27    >   INIT_FCALL_BY_NAME                                         'mysql_fetch_array'
        28        SEND_VAR_EX                                                !3
        29        DO_FCALL                                        0  $6
        30        ASSIGN                                             $5      !4, $6
        31      > JMPNZ                                                      $5, ->23
  62    32    >   INIT_FCALL_BY_NAME                                         'mysql_close'
        33        SEND_VAR_EX                                                !2
        34        DO_FCALL                                        0
  64    35      > RETURN                                                     1
```

Figure 2.3: Intermediate language for the example depicted in Figure 2.1.

are presented separated by commas. These two columns are the most important as they tell us the operation being executed and its parameters. The remaining columns merely specify further characteristics about it.

Columns *I* and *O* indicate if it is an I/O operation, and *fetch* whether the variable being accessed is global. The *return* column indicates in which variable the result of the statement is stored. Note that, in this IL, variables are represented by a number preceded by a *$*, *˜*, *->*, or *!*. All but the last correspond to auxiliary variables created for execution purposes. On the other hand, variables preceded by *!* correspond to user-defined variables that are automatically mapped by VLD to these symbols beforehand. For example, the PHP instruction of line 45, the entry point `$_GET[’UserData’]`, is interpreted as a composition of the first three statements. The result from the first statement (`FETCH_R _GET`) is temporarily stored in `$6`. This variable is used as an operand in the second statement (`FETCH_DIM_R $6, UserData`). Its result is, in turn, temporarily stored in *$5*, which is finally used in the third statement (`ASSIGN !0, $5`) to assign its value to `!0` (`!0` will represent the `$tainted` variable).

2.3 Deep Learning for Natural Language Processing

ML excels in problems that humans can solve intuitively but have difficulties formalizing. For instance, it is easy for a human to distinguish between a dog and a cat but it is hard to exhaustively list all the differences between the two animals. Because of that, these problems are harder for machines to tackle. In many ML models, such as Logistic Regression, data representation is preponderant for model performance - it is imperative to gather and select the appropriate features which will be used to represent each data instance. In the previous example, if we used the number of legs as a feature, we would probably not be able to distinguish the two species. Maybe a Boolean feature representing whether the animal has pointy ears would be more useful. For many tasks, it is hard to decide which representation to use though.

In DL models, instead of specifying the features, it is possible to learn them together with the main task. In our example we could simply use the pixels of pictures of dogs and cats, saving a lot of effort thinking which features would be better to distinguish the two animals. In these cases, DL models are good alternatives. DL models are constituted by multiple layers. Each layer receives as input the output of another layer and applies some additional transformation. By having multiple layers, the model can learn more complex data and patterns, based on simpler and broader ones (from previous layers) [20].

Like any ML model, DL models have a cost function they look to optimize, i.e., they are optimization tasks for which they need an optimization algorithm and a loss function (the target function to be optimized). Usually, neural network optimizers are based on stochastic gradient descent (SGD) [1]. This optimizer is a stochastic online version of the gradient descent, which iterativelly updates the weight matrix in the opposite direction of the gradient of the loss function.

Contrary to the traditional algorithm, SGD utilizes a single data point to perform the update, chosen at random. Equation 2.1 provides a formalization of the SGD, where W is the weight matrix, μ is the learning rate, and L_i is the loss with respect to the ith point of the dataset (where i is randomly chosen).

$$W \leftarrow W - \mu \frac{\partial L_i}{\partial W} \tag{2.1}$$

The learning rate is the hyperparameter responsible for dictating how big the update is. It is also directly related to how many updates the model takes to converge to a minimizer of the loss function. If the learning rate is small, the updates are minor, and it will take a long time for the model to converge. On the other hand, if it is too large, one expects the algorithm to converge in a fewer number of steps. This comes with the disadvantage that a better solution may be ‘missed’ by algorithm. Hence, the choice of the learning rate parameter must be taken wisely.

There are three SGD-based optimizers that are frequently used in DL models: 1) ADADELTA [56], 2) RMSProp [49], and 3) ADAM [25]. In ADADELTA, the learning rate is dynamic and the updates are made per dimension based on a moving window of gradient updates (instead of all past updates). This allows ADADELTA to continue learning even if many updates are done. RMSProp keeps a moving average of the square of past gradients. When updating the weight matrix, the gradient is divided by the square root of this average, which functions as a rescale of the current gradient. ADAM resorts to estimations of the first and second moments of the loss function instead of the gradient itself. Hence, the computation of ADAM is normally faster a lighter, turning it into a very appealing algorithm for DL models.

NLP deals with natural language data. Because of data characteristics, namely the lack of structure, ambiguity, discreteness, and sparseness [19], it is often hard to find a suitable representation for the desired task, making DL methods very popular in NLP. [37, 24]. There are two widely used components in DL for NLP: Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) [19]. These are not standalone layers but are important as feature extractors. CNN is a type of feed-forward network that can extract local features from the data. RNN architectures take into account both word ordering and past words in a sequence. They are excellent for sequential data and have achieved state-of-the-art results in many NLP fields [48, 5].

CNN and RNN layers are often preceded by an embedding layer, which maps discrete symbols into continuous vectors. This is a way of handling the sparsity problem that is common in NLP. It is also common to feed the output of these components to a feed-forward component that learns to perform the desired task, which in most cases is classification [19]. Recently, new approaches that combine different methods have obtained exciting results, such as the combination of attention and RNN or CNN [53, 33] and attention only, with resort to transformers [50, 15].

2.3.1 Convolutional Neural Networks

CNNs are a type of feed-forward network that, due to its architecture, is great at finding informative local patterns in long sequences with different sizes. The main idea behind CNN layers is to apply a non-linear learned function (filter) to a window of size k. At each time step, the window moves, until until it has covered the whole sequence, and produces a scalar value that represents the tokens from that time window [19]. We can apply n filters to each window, which results in an n dimensional vector that characterizes that window. The resulting vectors are combined through a *pooling operation* into a single vector, that represents the whole sequence. There are several pooling operations. The most common are:

- *Max Pooling*: takes the maximum value across each feature;
- *Average Pooling*: takes the average of each feature;
- *K-max Pooling*: for each feature, it keeps the k highest values, preserves their order, and concatenates the values into vectors.

In general, CNN cannot extract the global order of the input - only local order can be represented. Hence, this architecture is especially good in solving computer vision tasks, like image classification and object recognition [26, 45].

2.3.2 Recurrent Neural Networks

The first DL language models used a feed-forward network called tapped delay line (TDL). These networks receive as input the token at position t and the previous w tokens, where w is pre-determined. In [44], they train a TDL network to pronounce written English words. This approach, however, has a clear disadvantage. If w is too small, the model might miss interesting patterns, and if it is too long, it will be overloaded with parameters and may overfit. Besides, each token will be independently processed several times, in different time steps [8]. RNNs are networks that maintain a short-term memory through internal state space. The state space can be seen as a trace of previously processed input and enables the representation of dependencies between tokens that may be closer or further apart [8].

RNNs take into account all previous inputs in a more efficient fashion and without the trouble of tuning the hyperparameter w. There are several kinds of recurrent units. The simplest one is called Simple Recurrent Neural Network (SRNN) [16], and its internal state has a single recurrent layer that receives the output of the previous state and applies an activation function. Bengio et al. [7] noticed that even though SRNNs can learn short-term dependencies, long sequences led to vanishing or exploding gradients, making it difficult for the model to learn them.

The LSTM unit [23] was developed to tackle this issue. Figure 2.4 shows how the different parts of an LSTM unit (represented by the grey circles) work. LSTM models can maintain the error flow constant by introducing two gates - *input* and *forget* - that control how much information they let in [11]. The input gate i_t controls the input x_t, whereas the forget gate f_t controls the output of the previous unit, y_{t-1}. These gates produce a value between 0 and 1 (0 means they do not let anything pass, and 1 implies everything passes). Equations (2.2) show how to compute i_t and f_t. In equations i_t and f_t, respectively, U_i, W_i and b_i are the weight matrices and

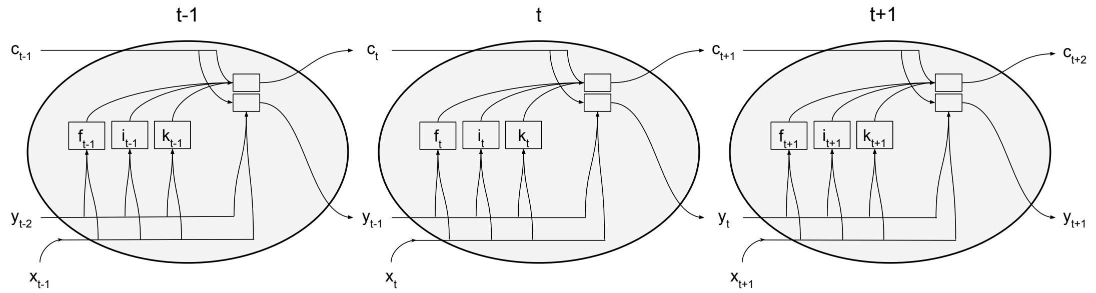


Figure 2.4: Scheme of an LSTM, showing the units $t-1$, t and $t+1$.

the bias for the input gate, and U_f, W_f and b_f are the weight matrices and bias for the forget gate.

$$\begin{aligned} i_t &= activation(dot(y_{t-1}, U_i) + dot(x_t, W_i) + b_i) \\ f_t &= activation(dot(y_{t-1}, U_f) + dot(x_t, W_f) + b_f) \end{aligned} \tag{2.2}$$

Each unit has a data flow c_t that carries information across time steps. Besides the gates, there is also a simple hidden layer component, k_t, whose equation is given by Equation (2.3), in which U_k, W_k, and b_k represent the weight matrices and bias for the hidden layer.

$$k_t = activation(dot(y_{t-1}, U_k) + dot(x_t, W_k) + b_k) \tag{2.3}$$

The next carry data flow c_{t+1} is computed by combining c_t, i_t, f_t and k_t, expressed by Equation (2.4):

$$c_{t+1} = i_t * k_t + c_t * f_t. \tag{2.4}$$

Finally, the output of the unit (and state of the next unit), is calculated by Equation (2.5), where c_t, x_t, and y_{t-1} are combined via a dense transformation. Analogously to Equations (2.2) and (2.3), here we also have weight matrices U_y, W_y and V_y, and a bias vector b_y.

$$y_t = activation(dot(y_{t-1}, U_y) + dot(x_t, W_y)) + dot(c_t, V_y) + b_y) \tag{2.5}$$

In the last few years, other recurrent units have appeared. The Gated Recurrent Unit (GRU) [10] is a more recent RNN very similar to the LSTM, but it was developed to be cheaper to run. It may, however, not have as much representational power as an LSTM layer [11]. The Bidirectional Recurrent Neural Network (BRNN) [43] can learn based not only in the past but in future information as well, widening the context considered. Figure 2.5 shows what happens inside a BiRNN layer. The BRNN feeds into an RNN the input vector and the reversed input vector to another RNN.

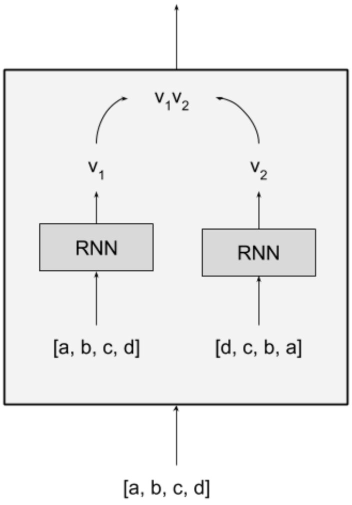


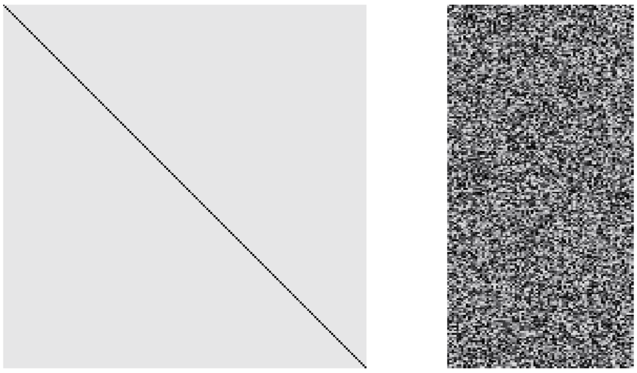
Figure 2.5: Scheme of a BRNN.

2.3.3 Input Representation

In NLP, it is necessary to represent text data as numeric vectors to be easily manipulated. The first thing to do is to decide the granularity of the representation: a vector may represent a sentence/sequence, a token (word or other character sequence separated by a space), a character, etc. Let us assume it represents a token. There are two main representation methods: one-hot vectors and embedding vectors. One-hot vectors are binary vectors where each entry is associated with a specific token, and it is either equal to 1 if the token condition is true, or 0 if it is false. This results in a binary sparse vector with dimension equal to the vocabulary (set of unique tokens in the dataset) size. This form of representation has been known to degrade the performance in neural network models [19]. On the other hand, embedding vectors are continuous representations in a lower-dimensional space. They can capture similarities between tokens, allowing the model to treat tokens with similar embedding representations in a similar way [19]. With embedding vectors, we can choose the size, and tune it to improve the model's performance.

The concatenation of the representations forms a matrix of parameters, that may be:

- *pre-trained*: there are specific models for embedding training (e.g., Word2vec [38] and GloVe [28]) that can be applied on a broader dataset. For instance, if we intend to classify English news as fake or not fake, we can use a large corpus of English documents from multiple areas to grasp better each word context, and then use the pre-trained embeddings on our model;
- *static* or *dynamic*: when dynamic embeddings are used, we allow the matrix



(a) One-hot encoding

(b) Embedding encoding

Figure 2.6: Heatmap for one-hot and embedding encodings of the Opcode Dataset vocabulary, where each line corresponds to the encoding of a token from the vocabulary.

of representations to change its values during training. This is always the case for embeddings that are not pre-trained. If we choose to pre-train the representations, however, we may want them to adapt to our data (and use the dynamic approach) or simply use them as they are (with the static approach).

Figure 2.6 shows two heatmaps representing the encodings of the vocabulary from the Opcode Dataset, which is one of the datasets we built. On the left there is the one-hot representation and the embedding representation is on the right. In both heatmaps, a horizontal ”line” represents a different token from the vocabulary. It is clear from the figure that the one-hot representation requires a higher dimensional space, as it always needs as many features as unique tokens (vocabulary size). In addition, it is a sparse representation - image (a) shows a large light grey area corresponding to values 0 and only the diagonal has black dots that correspond to values 1. The embedding image is dense and needs fewer features, since each feature is more meaningful. This also means the model will have fewer parameters, which helps prevent overfitting.

Word2vec is a popular model used to train word embeddings. There are two Word2vec approaches, the Continuous Bag-Of-Words (CBOW) model and the Continous Skip-Gram model. They are both composed of a projection layer (a simple linear layer). However, the CBOW model tries to predict the missing word given a context while the Skip-Gram tries to predict the context given a word [36]. Normally, the second model achieves better results and it is the one generally used. In Figure 2.7 there are the two models diagrams. As we can see, the CBOW model receives the context $x_{t-2}, x_{t-1}, x_{t+1}, x_{t+2}$, with window size equal to 2, of the word x_t and tries to predict it. On the other hand, the Skip-Gram model receives the word

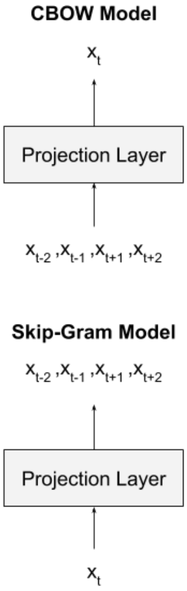

CBOW Model
x_t
Projection Layer
$x_{t-2}, x_{t-1}, x_{t+1}, x_{t+2}$
Skip-Gram Model
$x_{t-2}, x_{t-1}, x_{t+1}, x_{t+2}$
Projection Layer
x_t

Figure 2.7: CBOW and Skip-Gram model diagrams.

x_t as input and tries to predict its context. The main advantage of these models is that they are able to grasp the semantics of the words. For instance,

$$vec(''Madrid'') - vec(''Spain'') + vec(''France'') \simeq vec(''Paris''),$$

which shows the power of Word2vec in finding syntactic and semantic word relationships.

2.4 Machine Learning in Vulnerability Detection

ML entered the vulnerability detection field as a component in their models that automatizes some parts of the task. For instance, Yamaguchi et al. [54] use Principal Component Analysis (PCA) to create vector representations that describe API usage patterns in the code. Expert analysts then study the presence of vulnerabilities through these vectors and classify them. Another approach [34] applies taint analysis to PHP code to extract possible vulnerabilities. Next, they apply ML models to classify them as vulnerable or not. Here, the ML models help diminish false positives, a well-known problem of static analysis. Nevertheless, these models still require a great amount of specialized human effort.

In the following years, ML started to be used by a few authors to identify vulnerabilities. Medeiros et al. [35] extract code slices from PHP programs and translate them into an IL developed by the authors which are then classified by a Hidden Markov Model (HMM) to determine if they are vulnerable. However, a potential limitation of this approach is that the IL developed by the authors may loose insights about the computation behind it. Our IL based on PHP bytecode can decompose complex functions into simpler ones, exposing how they are executed and their interconnections inside the program. This choice deletes noise present in source code and at the same time introduces internal information that we believe is useful for vulnerability detection. Also, since they use an HMM, it was necessary to define manually the features that represent the PHP slices. In DL models, feature representation is greatly simplified.

Recently, DL gave its first steps in the field of vulnerability detection to enhance automation and reduce human experts' load [42, 21, 32, 31]. However, the vast majority focuses on the C/C++ languages [42, 32, 31, 30, 57], which have very different vulnerability characteristics and origins than those of the web. Since PHP is present in most web applications, it is imperative to investigate effective models that locate vulnerabilities in this language. To the best of our knowledge, there is no DL model that detects web flaws. Even recent DL models for C/C++, such as Devign [57], still present some issues: code and dataset are unavailable to the public, the results are questionable, considering the outdated tools that were used for comparison, and use function level granularity. The examples we use to train our model have program level granularity, where the instructions may come from different functions. Moreover, we are the first to explore vulnerability discovery at PHP bytecode level. Some works have used NLP techniques, such as Word2vec, to pre-train the embedding vectors [21, 32]. This allows input vectors to have semantic information embedded in them. Pre-training embedding vectors should be done in a large dataset, not necessarily the same used for the main task. Because it does not have to be labeled, it is usually easier to construct. This approach is quite relevant, especially when the available dataset is small.

There are two interesting approaches that leverage Lower Level Virtual Machine (LLVM) intermediate code for C/C++ programs [6, 30]. The first creates an embedding space, inst2vec, for code instructions in LLVM representing it as a graph that comprises both the Data Flow Graph (DFG) and the Control Flow Graph (CFG). The concepts of DFG and CFG are defined in the next section. The second starts by extracting the relevant lines from the LLVM code representation to refine the granularity. Then, they train a model composed of a standard BRNN configuration to which they added three pooling layers to deal with the fine-grained representation.

In the related field of malware detection, Guo et al. [22] developed a blackbox mixture model to interpret DL models. Although it is not in the scope of our work, it is an important subject to study in the future, since interpretability is essential in vulnerability detection and DL is quite hard to explain, which is often pointed as a reason to mistrust DL.

To the best of our knowledge, the way we address this task is new and has never been tried before. Previous work is either on other languages that do not suffer the same types of vulnerabilities [42, 32, 31, 29, 57], use methods that do not take into account the order of each token [21], or do not use an IL easily scalable and flexible [35].

2.5 Program Representation

A common limitation pointed out in past research is related to code representation. DL models for vulnerability detection frequently derive from NLP [42, 31, 32, 30, 6], where code is interpreted as a natural language. This abstraction does not consider the control and data flow of programs. Hence, a few works resort to traditional graph representations for programs, such as *Abstract Syntax Tree* (AST), *Data Flow Graph* (DFG), *Control Flow Graph* (CFG), *Program Dependency Graph* (PDG), and *Code Property Graph* (CPG), to detect vulnerabilities [4, 29, 57]. Let us briefly present each graph representation:

- *AST* [2] is an ordered tree representation whose purpose is to expose the abstract syntactic structure of a program in a certain programming language. Each node corresponds to a construct in the source code, which may be an operation (inner nodes) or operands (leaf nodes).

- *CFG* [3] is a directed binary tree where each node has two successors with attributes T (true) and F (false) associated with the outgoing edges. The subtree with the attribute T side is executed if the statement node represents is true, otherwise the program executes the subtree with attribute F.

- *DFG* [40] represents the global data dependence at the operator level and it was created for program optimization.

- *PDG* [17] is a program representation in which the nodes are statements and predicate expressions and the edges of a node represent both the data values on which the node's operations depend and the control conditions on which the execution of the operations depends. In other words, a PDG accommodates both the data and control dependencies.

- *CPG* [55] is a directed edge-labelled graph that comprises the AST, CFG and PDG of the program. Through the assignment of properties to both nodes and edges, we can know in which simpler representation we are in, which helps with traversals and information extraction. This representation results in a more complex graph where we can extract syntax, control and data dependencies.

Backes et al. [4] stores each PHP program as a CPG, to which they add a Call Graph - a directed graph connecting function call nodes that allow reason about control and data flows at an interprocedural level. Vulnerabilities are then discovered by appropriate graph traversals. Zhou et al. [57] represents a C function by a graph that accommodates the AST, CFG, DFG, and Natural Code Sequence. Thereafter, they train a Graph GRU that learns how to detect vulnerabilities from these representations.

One of the most recent papers in vulnerability detection for the C language [29] starts by creating a System Dependency Graph, a PDG to which they add the inter-procedural invocations. From this representation they extract a reduced set of statements which relate to a security flaw. This work follows a granularity strategy similar to ours, by extracting a sequence of tokens from an inter-procedural program excerpt that takes into consideration a security-critical operation. In our work, we also start with PHP excerpts that end in a security-critical operation (sensitive sink). However, we will not use the source code directly, but the bytecode.

These approaches create an overly complicated representation of the code, which could be difficult to learn and generalize. We believe that our approach with an IL for PHP is cleaner, and can still accommodate the necessary program dependencies without resorting to these complex structures.

Chapter 3

Datasets and Deep Learning Architectures for SQLi Detection

Chapter 3 presents our solution to detect SQLi vulnerabilities by leveraging DL, as well as the definition of the problem that it solves and the datasets we built to evaluate it. In fact, the chapter presents the three main contributions, which correspond to the following sections:

1. Problem Definition (Section 3.1)

2. SQLi Detection Deep Learning Network (Section 3.2)

3. Datasets (Section 3.3).

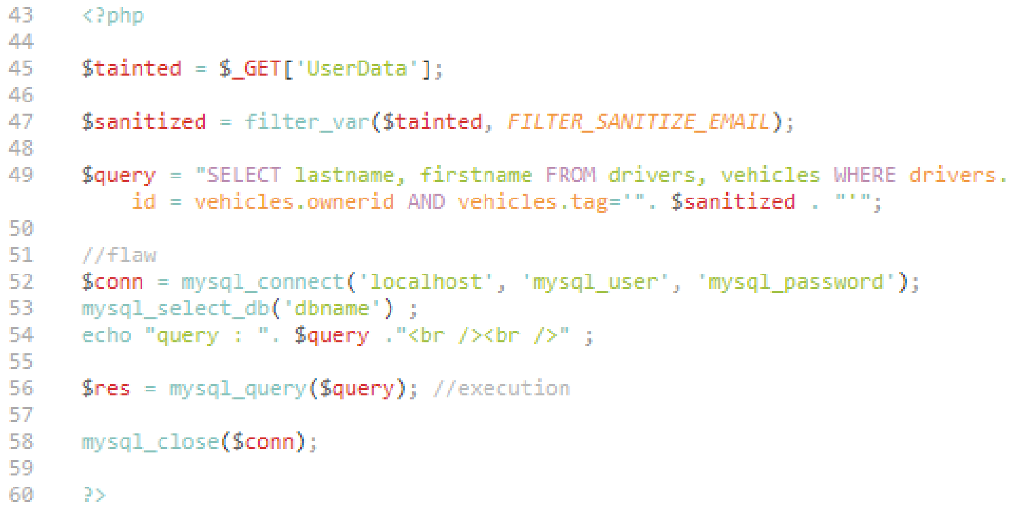
In Section 3.1, we define the problem of classifying PHP code represented in an intermediate language as vulnerable or not to SQLi. This section is the lead-off of the thesis, which makes it of the uttermost importance. Here, the reader can grasp better the nuances we are faced with when solving this problem.

Section 3.2 presents the solution we built for the problem. We designed a DL network which, through the different layer types that compose it, predicts how likely the PHP code is of being vulnerable to SQLi.

Finally, Section 3.3 defines the datasets we used. We built six datasets to evaluate which one is a better fit to our model, namely, the PHP Excerpt Dataset, the Bytecode Excerpt Dataset, the Opcode Dataset, the Opcode + Operand Dataset, the Slice Dataset, and the Simplified Slice Dataset. The first contains raw PHP excerpts retrieved from the SARD. The second contains the corresponding excerpts in the IL. The remaining datasets were built iteratively with different information from the IL and were used to train the network.

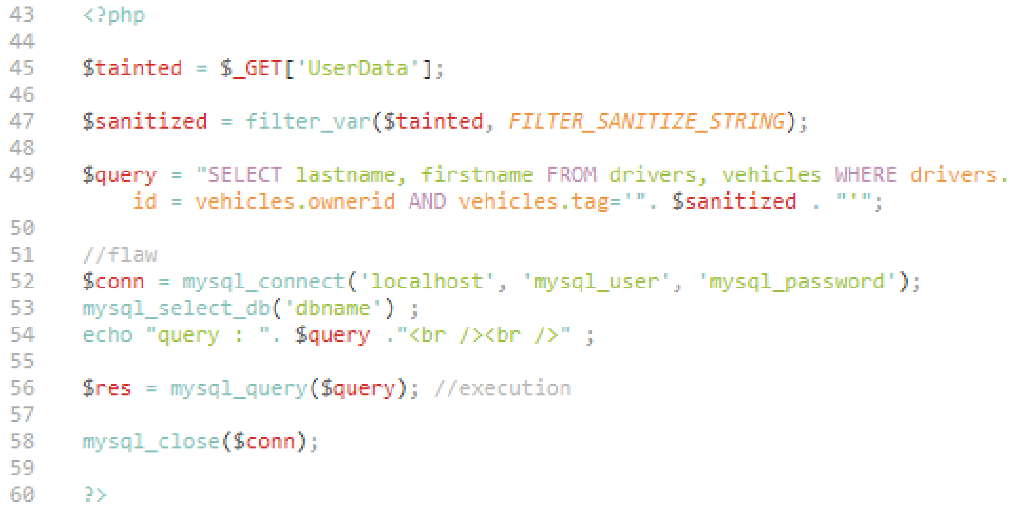
3.1 Problem Definition

In this section we present the detection of SQLi vulnerabilities in PHP source code, by analyzing it through an IL and resorting to a DL model. Hence, the section is divided into three subsections. The first is the main description of the problem. We will show different examples of code with and without SQLi vulnerabilities and how it reflects in the corresponding IL. Then, we will present two subproblems that arise from it: 1) which data is most suitable to solve the problem, and 2) which model can better classify the data samples correctly.



```
<?php

$tainted = $_GET['UserData'];

$sanitized = filter_var($tainted, FILTER_SANITIZE_EMAIL);

$query = "SELECT lastname, firstname FROM drivers, vehicles WHERE drivers.
    id = vehicles.ownerid AND vehicles.tag='". $sanitized . "'";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

mysql_close($conn);

?>
```

Figure 3.1: Example of a vulnerable code excerpt adapted from a SARD sample.



```
<?php

$tainted = $_GET['UserData'];

$sanitized = filter_var($tainted, FILTER_SANITIZE_STRING);

$query = "SELECT lastname, firstname FROM drivers, vehicles WHERE drivers.
    id = vehicles.ownerid AND vehicles.tag='". $sanitized . "'";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
mysql_select_db('dbname') ;
echo "query : ". $query ."<br /><br />" ;

$res = mysql_query($query); //execution

mysql_close($conn);

?>
```

Figure 3.2: Example of a non-vulnerable code excerpt adapted from a SARD sample. Note the similarities of this code excerpt and Figure 3.1

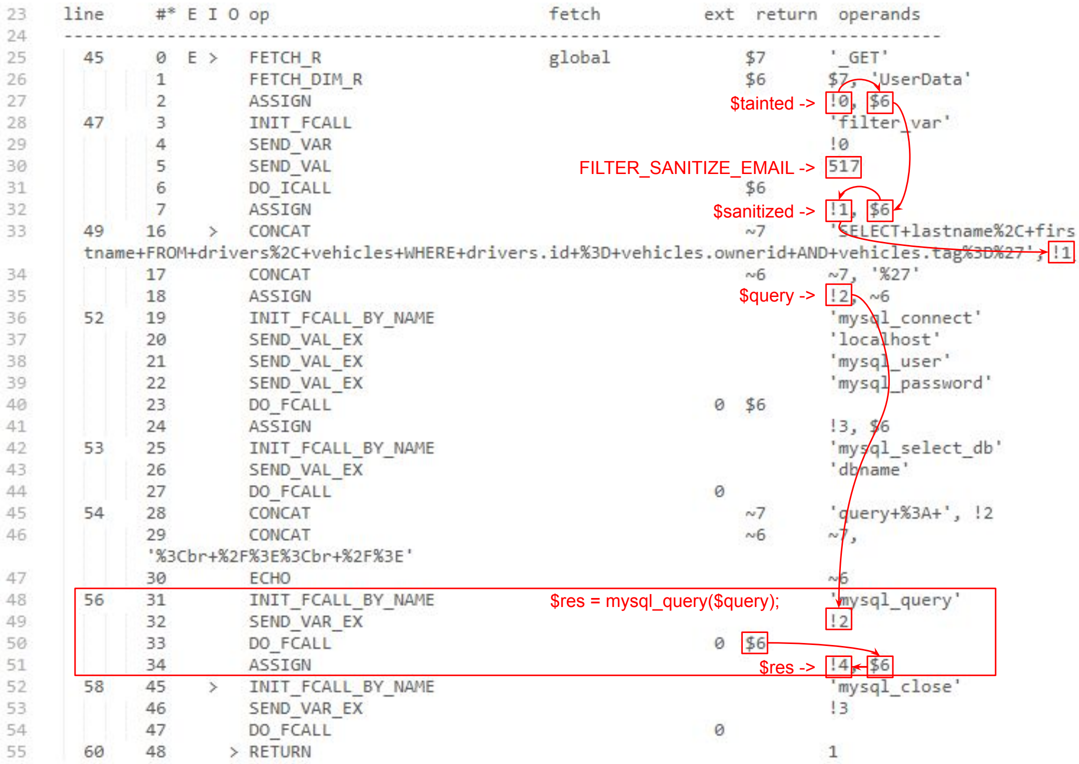
Suppose there is a PHP application to which we can access the source code to check if it contains SQLi vulnerabilities. Our goal is to create a framework to detect SQLi vulnerabilities by analyzing its IL through a DL model. Typically, web applications have more than one file, and the vulnerability may be scattered across multiple files. Besides, not all lines of code are equally relevant to detect SLQi. Hence, choosing the right granularity to solve this task is the first important decision, as it dictates how useful the model will be in practice. For example, models that only detect intra-function vulnerabilities, or intra-file vulnerabilities, end up missing critical cases. If the input is sanitized in a function that belongs to a different file from where the entry point is located, those models will miss it.

We will consider as analysis unities *code excerpts* in an IL. A code excerpt is a sequence of lines (*instructions*) that begins in an entry point and ends in a sensitive sink. Those lines do not necessarily belong to the same file, though. This approach prevents the model from missing cases as the one explained above. Each code excerpt can be represented in the IL by a sequence of *statements*, which comprises a *bytecode excerpt*.

Therefore, the problem can be defined as **finding the best DL model (in a certain sense) that takes a bytecode excerpt as input, and decides whether the excerpt is vulnerable to SQLi or not.** Ideally, the model achieves this by learning some structure from several bytecode excerpts to which we give the label in advance (known as *supervised learning*). Even though we will use DL, in which a lot of reasoning is done for us by the model, understanding how an SQLi is detected is key to finding better solutions. So, let us consider the code excerpts depicted in Figures 3.1 and 3.2.

The two excerpts are pretty similar. They start by storing some input through the global variable `$_GET` (line 45), which is then sanitized in respect to a filter (line 47). Next, the result is concatenated to a query string, which is then executed (lines 49 and 56). In Figure 3.1, the filter applied is the `FILTER_SANITIZE_EMAIL`, which aims to check if a given string follows the email format (e.g., *mail@mail.com*). However, it cannot escape all metacharacters from SQL, such as *'* and –, which makes it possible for an attacker to inject malicious SQL code in the input. This excerpt is, therefore, vulnerable to SQLi. On the other hand, `FILTER_SANITIZE_STRING` can escape all SQL metacharacters. So, Figure 3.2 is not vulnerable to SQLi.

Figures 3.3 and 3.4 show the bytecode excerpts for the excerpts of Figures 3.1 and 3.2, respectively. Here, we can see that there are important aspects that help us in classifying the excerpts. Firstly, there are the dataflows from the input (stored in `!0`) and the query string (stored in `!2` (line 18), which can be followed through the operands' column. Then, it is important to identify which PHP functions are being executed. For example, the excerpt 3.3 would not be vulnerable if the `mysql_query` function was not called on the query string (query execution in lines 31-32 of the figure). Without it, the attacker would not have access to the information in the



```
line     #* E I O op                           fetch          ext  return  operands
-------------------------------------------------------------------------------------
  45     0  E >   FETCH_R                      global              $7      '_GET'
         1        FETCH_DIM_R                                      $6      $7, 'UserData'
         2        ASSIGN                                                   !0, $6
  47     3        INIT_FCALL                                               'filter_var'
         4        SEND_VAR                                                 !0
         5        SEND_VAL                                                 517
         6        DO_ICALL                                         $6
         7        ASSIGN                                                   !1, $6
  49    16      > CONCAT                                           ~7      'SELECT+lastname%2C+firs
tname+FROM+drivers%2C+vehicles+WHERE+drivers.id+%3D+vehicles.ownerid+AND+vehicles.tag%3D%27', !1
        17        CONCAT                                           ~6      ~7, '%27'
        18        ASSIGN                                                   !2, ~6
  52    19        INIT_FCALL_BY_NAME                                       'mysql_connect'
        20        SEND_VAL_EX                                              'localhost'
        21        SEND_VAL_EX                                              'mysql_user'
        22        SEND_VAL_EX                                              'mysql_password'
        23        DO_FCALL                                      0  $6
        24        ASSIGN                                                   !3, $6
  53    25        INIT_FCALL_BY_NAME                                       'mysql_select_db'
        26        SEND_VAL_EX                                              'dbname'
        27        DO_FCALL                                      0
  54    28        CONCAT                                           ~7      'query+%3A+', !2
        29        CONCAT                                           ~6      ~7,
'%3Cbr+%2F%3E%3Cbr+%2F%3E'
        30        ECHO                                                     ~6
  56    31        INIT_FCALL_BY_NAME                                       'mysql_query'
        32        SEND_VAR_EX                                              !2
        33        DO_FCALL                                      0  $6
        34        ASSIGN                                                   !4, $6
  58    45      > INIT_FCALL_BY_NAME                                       'mysql_close'
        46        SEND_VAR_EX                                              !3
        47        DO_FCALL                                      0
  60    48      > RETURN                                                   1
```

Figure 3.3: Bytecode excerpt corresponding to code excerpt in Figure 3.1

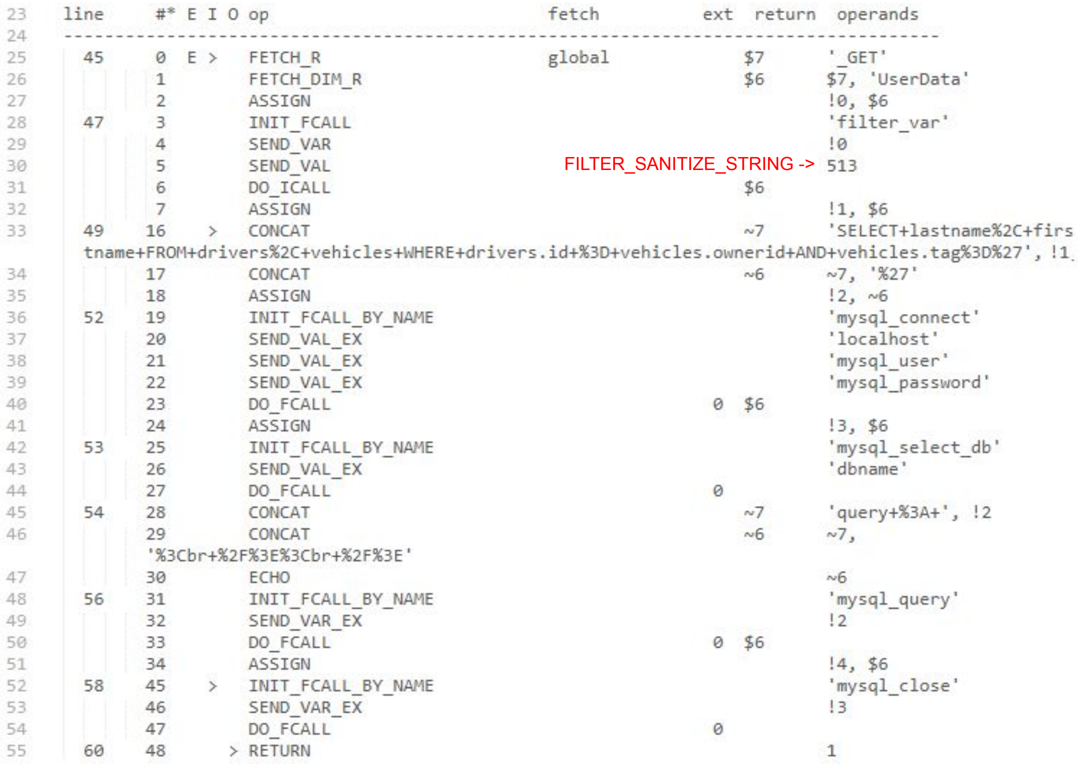
database. Another crucial information, which is the preponderant factor in this example, is the filter code, the numeric operand in the fifth statement (SEND_VAL 517 for the vulnerable sample). To identify it as the filter code, it has to be the operand of the SEND_VAL opcode and first statement of that instruction (line 47) has to be INIT_FCALL 'filter_var'.

This problem introduces two subproblems: which information to use to express vulnerabilities and semantic aspects of PHP, and which model is capable of processing such information to detect SQLi accurately. The following subsections offer some further details on why these problems exist and how they can be solved.

3.1.1 Which data to use?

As we discussed above, the first important decision we needed to take regarding data was related to the granularity of the samples analyzed. We saw that a good choice can be excerpts. Another important aspect is related to which information to choose from the bytecode excerpts. We could encode everything included, but it could turn out to be too much information to model. To balance overfitting and accuracy, one should opt to use the minimum information possible. So, which information is this?

A first approach might be to consider the opcodes only. The downside is that



```
line     #* E I O op                           fetch          ext  return  operands
-------------------------------------------------------------------------------------
  45     0  E >   FETCH_R                      global              $7      '_GET'
         1        FETCH_DIM_R                                      $6      $7, 'UserData'
         2        ASSIGN                                                   !0, $6
  47     3        INIT_FCALL                                               'filter_var'
         4        SEND_VAR                                                 !0
         5        SEND_VAL                     FILTER_SANITIZE_STRING -> 513
         6        DO_ICALL                                         $6
         7        ASSIGN                                                   !1, $6
  49    16    >   CONCAT                                           ~7      'SELECT+lastname%2C+firs
tname+FROM+drivers%2C+vehicles+WHERE+drivers.id+%3D+vehicles.ownerid+AND+vehicles.tag%3D%27', !1
        17        CONCAT                                           ~6      ~7, '%27'
        18        ASSIGN                                                   !2, ~6
  52    19        INIT_FCALL_BY_NAME                                       'mysql_connect'
        20        SEND_VAL_EX                                              'localhost'
        21        SEND_VAL_EX                                              'mysql_user'
        22        SEND_VAL_EX                                              'mysql_password'
        23        DO_FCALL                                      0  $6
        24        ASSIGN                                                   !3, $6
  53    25        INIT_FCALL_BY_NAME                                       'mysql_select_db'
        26        SEND_VAL_EX                                              'dbname'
        27        DO_FCALL                                      0
  54    28        CONCAT                                           ~7      'query+%3A+', !2
        29        CONCAT                                           ~6      ~7,
        '%3Cbr+%2F%3E%3Cbr+%2F%3E'
        30        ECHO                                                     ~6
  56    31        INIT_FCALL_BY_NAME                                       'mysql_query'
        32        SEND_VAR_EX                                              !2
        33        DO_FCALL                                      0  $6
        34        ASSIGN                                                   !4, $6
  58    45    >   INIT_FCALL_BY_NAME                                       'mysql_close'
        46        SEND_VAR_EX                                              !3
        47        DO_FCALL                                      0
  60    48      > RETURN                                                   1
```

Figure 3.4: Bytecode excerpt corresponding to code excerpt in Figure 3.2

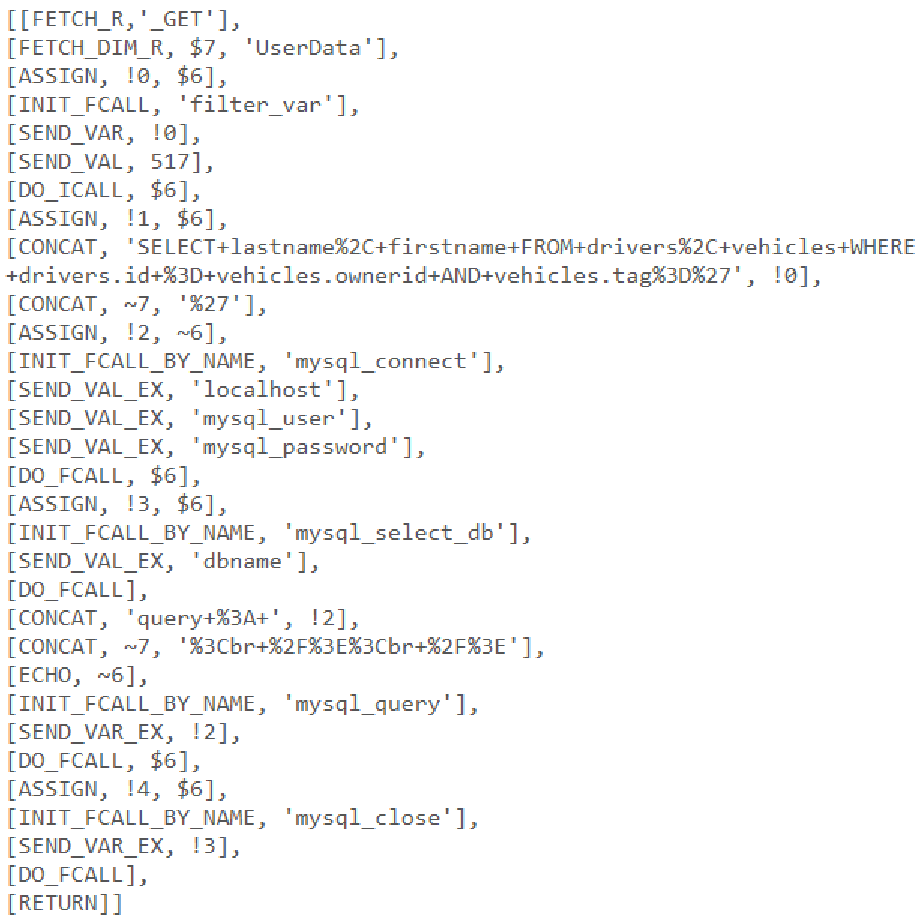
we would lose crucial information, such as the function names or their arguments. For instance, the two excerpts we showed before would be represented by the same opcode sequence (Figure 3.5). In this case, the model would be unable to learn the correct labels.

```
[FETCH_R, FETCH_DIM_R, ASSIGN, INIT_FCALL, SEND_VAR, SEND_VAL,
DO_ICALL, ASSIGN, CONCAT, CONCAT, ASSIGN, INIT_FCALL_BY_NAME,
SEND_VAL_EX, SEND_VAL_EX, SEND_VAL_EX, DO_FCALL, ASSIGN,
INIT_FCALL_BY_NAME, SEND_VAL_EX, DO_FCALL, CONCAT, CONCAT, ECHO,
INIT_FCALL_BY_NAME, SEND_VAR_EX, DO_FCALL, ASSIGN,
INIT_FCALL_BY_NAME, SEND_VAR_EX, DO_FCALL, RETURN]
```

Figure 3.5: Opcode sequence for the excerpts in Figures 3.3 and 3.4.

Thus, including the operands is also essential. In our examples, the sequences would no longer be equal, as they would contain the filter code, which is the key to determine the label. Figures 3.6 and 3.7 show the sequence of opcodes and operands for the vulnerable and non-vulnerable bytecode excerpts, respectively. As it is possible to observe, the sixth line on both sequences is different, meaning they can be distinguished.

When processed as natural data, these approaches pose a problem already exposed in Section 2.5. Contrary to natural data, programs are not necessarily pro-



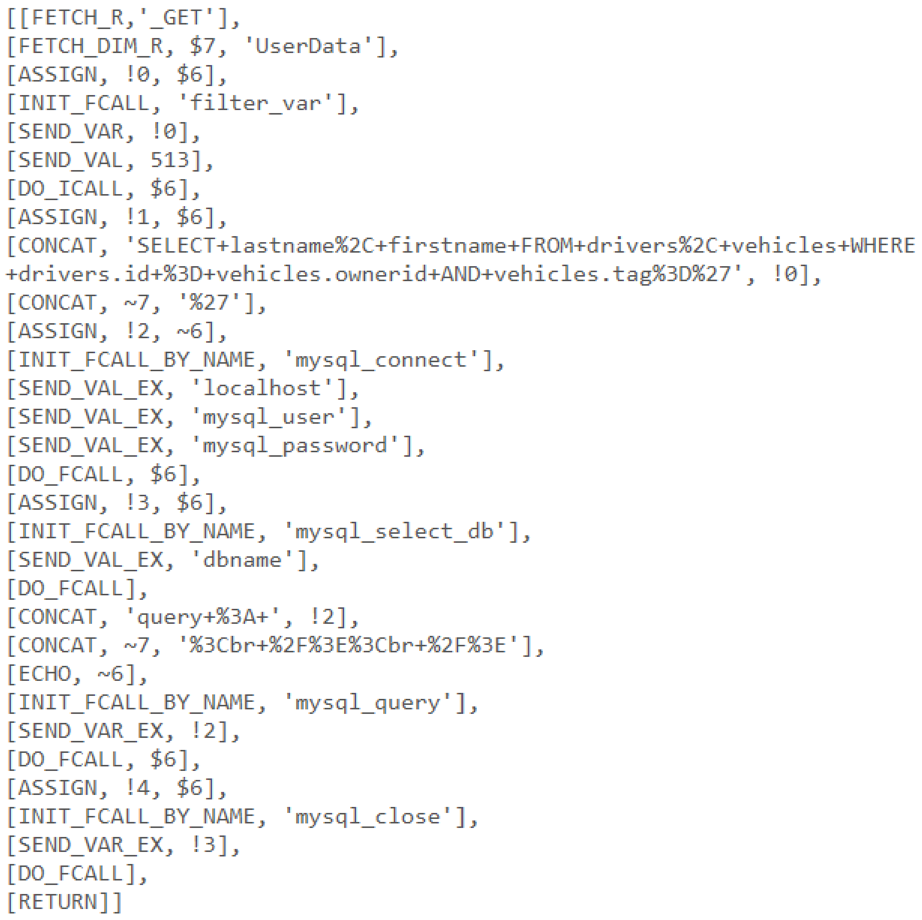
```
[[FETCH_R,'_GET'],
[FETCH_DIM_R, $7, 'UserData'],
[ASSIGN, !0, $6],
[INIT_FCALL, 'filter_var'],
[SEND_VAR, !0],
[SEND_VAL, 517],
[DO_ICALL, $6],
[ASSIGN, !1, $6],
[CONCAT, 'SELECT+lastname%2C+firstname+FROM+drivers%2C+vehicles+WHERE
+drivers.id+%3D+vehicles.ownerid+AND+vehicles.tag%3D%27', !0],
[CONCAT, ~7, '%27'],
[ASSIGN, !2, ~6],
[INIT_FCALL_BY_NAME, 'mysql_connect'],
[SEND_VAL_EX, 'localhost'],
[SEND_VAL_EX, 'mysql_user'],
[SEND_VAL_EX, 'mysql_password'],
[DO_FCALL, $6],
[ASSIGN, !3, $6],
[INIT_FCALL_BY_NAME, 'mysql_select_db'],
[SEND_VAL_EX, 'dbname'],
[DO_FCALL],
[CONCAT, 'query+%3A+', !2],
[CONCAT, ~7, '%3Cbr+%2F%3E%3Cbr+%2F%3E'],
[ECHO, ~6],
[INIT_FCALL_BY_NAME, 'mysql_query'],
[SEND_VAR_EX, !2],
[DO_FCALL, $6],
[ASSIGN, !4, $6],
[INIT_FCALL_BY_NAME, 'mysql_close'],
[SEND_VAR_EX, !3],
[DO_FCALL],
[RETURN]]
```

Figure 3.6: Opcode and operand sequence for the excerpts in Figure 3.3.

cessed sequentially. They might have user function calls, cycles, etc., which alters the control flow. Past work used graph program representations to tackle this issue, even when they increase the complexity. Another option can be to analyze the multiple execution paths from each excerpt. We will call it **slices**. For instance, the excerpt from Figure 3.8, which is an adaptation of the vulnerable sample from SARD, has two different slices: a vulnerable slice with the *true* branch of the if-statement, which is composed by lines {45, 47, 48, 49, 53, 56, 57, 58, 60, 62}; and a correct slice with the *false* branch, which is composed by lines {45, 47, 48, 50, 51, 53, 56, 57, 58, 60, 62}. Note that, even though the original excerpt is vulnerable, the second slice is non-vulnerable: the variable `$tainted` defined on line 45 with the user input is substituted by an empty string on line 51. This prevents the attacker to inject SQL code, but also invalidates the usage of any user input.

3.1.2 Which model to define?

The model and the data characteristics are closely related. They are, in fact, dependent on each other. As discussed in Section 2.3, the employment of DL models



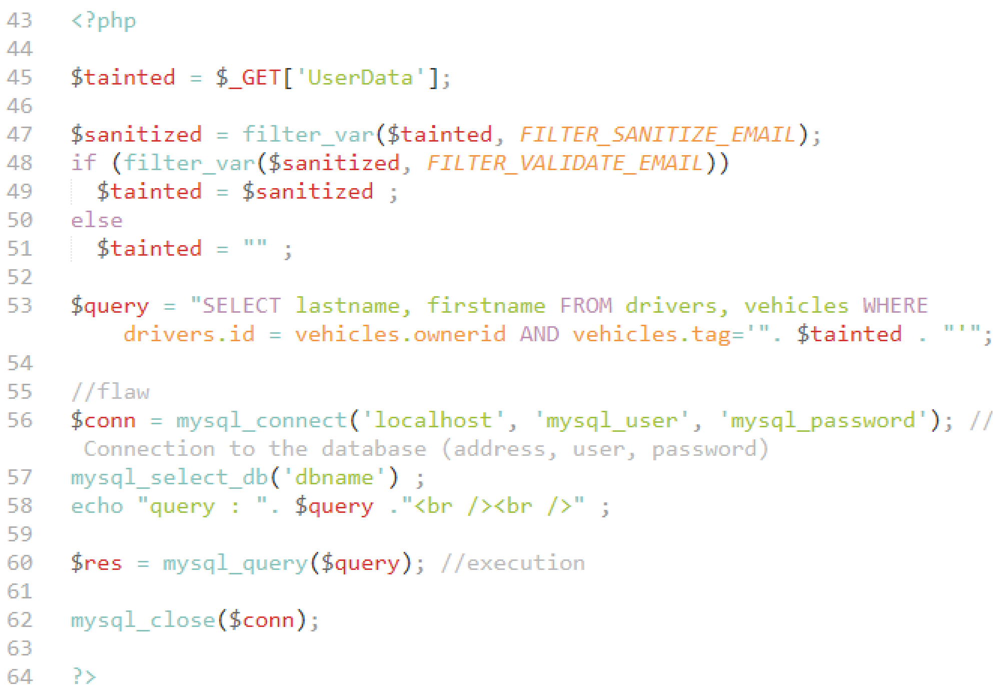
```
[[FETCH_R,'_GET'],
[FETCH_DIM_R, $7, 'UserData'],
[ASSIGN, !0, $6],
[INIT_FCALL, 'filter_var'],
[SEND_VAR, !0],
[SEND_VAL, 513],
[DO_ICALL, $6],
[ASSIGN, !1, $6],
[CONCAT, 'SELECT+lastname%2C+firstname+FROM+drivers%2C+vehicles+WHERE
+drivers.id+%3D+vehicles.ownerid+AND+vehicles.tag%3D%27', !0],
[CONCAT, ~7, '%27'],
[ASSIGN, !2, ~6],
[INIT_FCALL_BY_NAME, 'mysql_connect'],
[SEND_VAL_EX, 'localhost'],
[SEND_VAL_EX, 'mysql_user'],
[SEND_VAL_EX, 'mysql_password'],
[DO_FCALL, $6],
[ASSIGN, !3, $6],
[INIT_FCALL_BY_NAME, 'mysql_select_db'],
[SEND_VAL_EX, 'dbname'],
[DO_FCALL],
[CONCAT, 'query+%3A+', !2],
[CONCAT, ~7, '%3Cbr+%2F%3E%3Cbr+%2F%3E'],
[ECHO, ~6],
[INIT_FCALL_BY_NAME, 'mysql_query'],
[SEND_VAR_EX, !2],
[DO_FCALL, $6],
[ASSIGN, !4, $6],
[INIT_FCALL_BY_NAME, 'mysql_close'],
[SEND_VAR_EX, !3],
[DO_FCALL],
[RETURN]]
```

Figure 3.7: Opcode and operand sequence for the excerpts in Figure 3.4.

facilitates defining the characteristics for each sample since the layers can learn them along with the labels. This is especially useful with unstructured data such as natural data, code or bytecode excerpts.

Hence, it is necessary to choose the network configuration accordingly: types of layers and their sizes, number of layers, optimization function, and optimization algorithm. Some choices can be made promptly through reasoning (e.g., the choice of the optimization function). However, some require experimental work, like choosing the size of the network's layers or the number of layers. Finally, others require a bit of both, such as choosing the type of layers. We may start by a more straightforward configuration that fits the problem well, and then try to improve it through experimentation. It is through experimentation that the data and the model become dependent on each other. Nonetheless, it should not be too dependent but only enough to learn the general patterns relevant to solve the problem in question.

In our problem, we have unstructured data, the bytecode excerpts. We want to classify them as vulnerable or non-vulnerable to SQLi (i.e., *binary classification*). Since we have access to the excerpts' labels, the problem is known as *supervised learning*. Hence, we aim at finding a network configuration that can deal with



```php
<?php

$tainted = $_GET['UserData'];

$sanitized = filter_var($tainted, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized, FILTER_VALIDATE_EMAIL))
  $tainted = $sanitized ;
else
  $tainted = "" ;

$query = "SELECT lastname, firstname FROM drivers, vehicles WHERE drivers.id = vehicles.ownerid AND vehicles.tag='". $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

mysql_close($conn);

?>
```

Figure 3.8: Opcode and operand sequence for the excerpts in Figure 3.3.

unstructured data, in which the order is important to classify the sample. In Section 2.3 we looked into such layers: RNN and CNN layers. Both components may be used, however, since the related elements in the bytecode may be further away, an RNN layer seems a better fit to start with. Regarding the data used for training, we built different representations from bytecode sequences. In the remaining sections of this chapter, we present in detail the model and datasets we built to solve this problem.

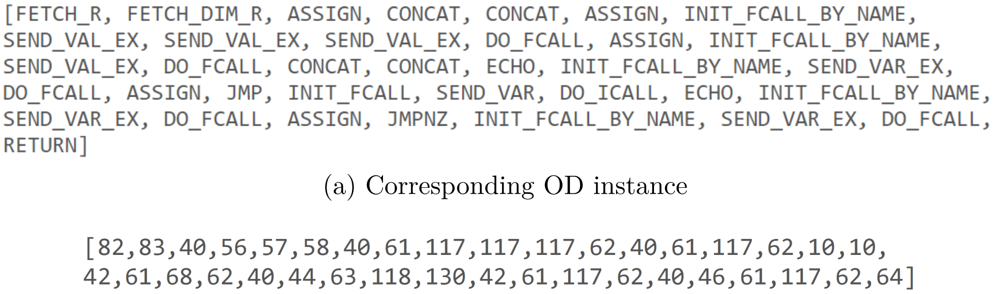
3.2 SQLi Detection Network

In this section, we present the methodology we use to classify PHP code (excerpts or slices) as vulnerable or non-vulnerable to SQLi, by processing them in an IL, i.e., bytecode. Firstly, we introduce a sequential DL network capable of classifying PHP code as being vulnerable or not. The network is composed of Embedding, LSTM, Dropout, and Dense layers, following the guidelines presented in Section 2.3. We then present the preprocessing steps the datasets were subject to, such that they can adequately serve as input to the network. Since the network can only deal with numeric vectors, we will explain how they are constructed from vectors of opcodes.

3.2.1 Preprocessing

Since neural networks receive as input arrays of numerical values, we created a numeric vector for each instance, by mapping the tokens to their corresponding index in the vocabulary. We will use one of the datasets we constructed, the Opcode Dataset (OD), to exemplify this procedure. This dataset is solely composed of the operation names in bytecode (opcodes). Table 3.1 lists OD's vocabulary. The index of the opcode in position (i, j) (row, column) is given by expression $i * 4 + j + 1$.

Figure 3.9 shows how an OD example (part (a)) is represented in a numeric vector (part (b)). As we can observe the first opcode of the example, *FETCH_R*, has index 82 (see Table 3.1 by applying the expression $20 \times 4 + 1 + 1 = 82$), which is the first element of the vector.



```
[FETCH_R, FETCH_DIM_R, ASSIGN, CONCAT, CONCAT, ASSIGN, INIT_FCALL_BY_NAME,
SEND_VAL_EX, SEND_VAL_EX, SEND_VAL_EX, DO_FCALL, ASSIGN, INIT_FCALL_BY_NAME,
SEND_VAL_EX, DO_FCALL, CONCAT, CONCAT, ECHO, INIT_FCALL_BY_NAME, SEND_VAR_EX,
DO_FCALL, ASSIGN, JMP, INIT_FCALL, SEND_VAR, DO_ICALL, ECHO, INIT_FCALL_BY_NAME,
SEND_VAR_EX, DO_FCALL, ASSIGN, JMPNZ, INIT_FCALL_BY_NAME, SEND_VAR_EX, DO_FCALL,
RETURN]
```

(a) Corresponding OD instance

```
[82,83,40,56,57,58,40,61,117,117,117,62,40,61,117,62,10,10,
42,61,68,62,40,44,63,118,130,42,61,117,62,40,46,61,117,62,64]
```

(b) Resulting numeric vector

Figure 3.9: Example of the resulting numerical vector from a sequence of opcodes.

LSTM layers do not support different sized inputs. For that reason, before training, we pad all smaller sequences with 0's at the end so that all sequences have the same size as the longest sequence in the training set. If a longer sequence appears when evaluating the model, then we truncate it. This means it is possible that the model may evaluate sequences that are incomplete, namely, without the last tokens. Even though this technique may affect the model's performance negatively, we consider it a fair approach since it is unlikely it will happen often.

Table 3.1: Vocabulary composed of VLD opcodes. The index of the opcode in position i, j is given by $i * 4 + j + 1$.

i \j	0	1	2	3
0	OOV	NOP	ADD	SUB
1	MULT	DIV	MOD	SL
2	SR	CONCAT	BW_OR	BW_AND
3	BW_XOR	BW_NOT	BOOL_NOT	BOOL_XOR
4	IS_IDENTICAL	IS_NOT_IDENTICAL	IS_EQUAL	IS_NOT_EQUAL
5	IS_SMALLER	IS_SMALLER_OR_EQUAL	CAST	QM_ASSIGN
6	ASSIGN_ADD	ASSIGN_SUB	ASSIGN_MUL	ASSIGN_DIV
7	ASSIGN_MOD	ASSIGN_SL	ASSIGN_SR	ASSIGN_CONCAT
8	ASSIGN_BW_OR	ASSIGN_BW_AND	ASSIGN_BW_XOR	PRE_INC
9	PRE_DEC	POST_INC	POST_DEC	ASSIGN
10	ASSIGN_REF	ECHO	GENERATOR_CREATE	JMP
11	JMPZ	JMPNZ	JMPZNZ	JMPZ_EX
12	JMPNZ_EX	CASE	CHECK_VAR	SEND_VAR_NO_REF_EX
13	MAKE_REF	BOOL	FAST_CONCAT	ROPE_INIT
14	ROPE_ADD	ROPE_END	BEGIN_SILENCE	END_SILENCE
15	INIT_FCALL_BY_NAME	DO_FCALL	INIT_FCALL	RETURN
16	RECV	RECV_INIT	SEND_VAL	SEND_VAR_EX
17	SEND_REF	NEW	INIT_NS_FCALL_BY_NAME	FREE
18	INIT_ARRAY	ADD_ARRAY_ELEMENT	INCLUDE_OR_EVAL	UNSET_VAR
19	UNSET_DIM	UNSET_OBJ	FE_RESET_R	FE_FETCH_R
20	EXIT	FETCH_R	FETCH_DIM_R	FETCH_OBJ_R
21	FETCH_W	FETCH_DIM_W	FETCH_OBJ_W	FETCH_RW
22	FETCH_DIM_RW	FETCH_OBJ_RW	FETCH_IS	FETCH_DIM_IS
23	FETCH_OBJ_IS	FETCH_FUNC_ARG	FETCH_DIM_FUNC_ARG	FETCH_OBJ_FUNC_ARG
24	FETCH_UNSET	FETCH_DIM_UNSET	FETCH_OBJ_UNSET	FETCH_LIST
25	FETCH_CONSTANT	GOTO	EXIT_STMT	EXT_FCALL_BEGIN
26	EXT_FCALL_END	EXT_NOP	TICKS	SEND_VAR_NO_REF
27	CATCH	THROW	FETCH_CLASS	CLONE
28	RETURN_BY_REF	INIT_METHOD_CALL	INIT_STATIC_METHOD_CALL	ISSET_ISEMTY_VAR
29	ISSET_ISEMPTY_DIM_OBJ	SEND_VAL_EX	SEND_VAR	INIT_USER_CALL
30	UNKNOWN[119]	SEND_USER	STRLEN	DEFINED
31	TYPE_CHECK	VERIFY_RETURN_TYPE	FE_RESET_RW	FE_FETCH_RW
32	FE_FREE	INIT_DYNAMIC_CALL	DO_ICALL	DO_UCALL
33	DO_FCALL_BY_NAME	PRE_INC_OBJ	PRE_DEC_OBJ	POST_INC_OBJ
34	POST_DEC_OBJ	ASSIGN_OBJ	OP_DATA	INSTANCEOF
35	DECLARE_CLASS	DECLARE_INHERITED_CLASS	DECLARE_FUNCTION	RAISE_ABSTRACT_ERROR
36	DECLARE_CONST	ADD_INTERFACE	VERIFY_INSTANCEOF	VERIFY_ABSTRACT_CLASS
37	ASSIGN_DIM	ISSET_ISEMPTY_PROP_OBJ	HANDLE_EXCEPTION	USER_OPCODE
38	ASSERT_CHECK	JMP_SET	DECLARE_LAMBDA_FUNCTION	ADD_TRAIT
39	BINDTRAIS	SEPARATE	FETCH_CLASS_NAME	JMP_SET_VAR
32	DISCARD_EXCEPTION	YIELD	GENERATOR_RETURN	FAST_CALL
33	FAST_RET	RECV_VARIADIC	SEND_UNPACK	POW
34	ASSIGN_POW	BIND_GLOBAL	COALESCE	SPACESHIP
35	DECLARE_ANON_CLASS	DECLARE_ANON_INHERITED_CLASS	FETCH_STATIC_PROP_R	FETCH_STATIC_PROP_W
36	FETCH_STATIC_PROP_RW	FETCH_STATIC_PROP_IS	FETCH_STATIC_PROP_FUC_ARG	FETCH_STATIC_PROP_UNSET
37	UNSET_STATIC_PROP	ISSET_ISEMPTY_STATIC_PROP	FETCH_CLASSICAL_CONSTANT	BIND_LEXICAL
38	BIND_STATIC	FETCH_THIS	UNKNOWN[185]	ISSET_ISEMPTY_THIS
39	SWITCH_LONG	SWITCH_STRING	IN_ARRAY	COUNT
39	GET_CLASS	GET_CALLED_CLASS	GET_TYPE	FUNC_NUM_ARGS
39	FUNC_GET_ARGS	ISSET_EMPTY		

3.2.2 Deep Learning Model

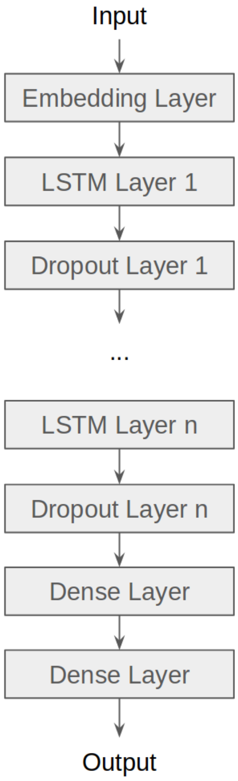


Figure 3.10: High level overview of our model.

To address the SQLi vulnerability detection, we propose a DL model following the guidelines presented in Section 2.3. Figure 3.10 gives a high-level overview of our network. The network is composed of a minimum of five layers that work sequentially. It produces a final output, between 0 and 1, indicating the probability of the sample being vulnerable. It receives as input a numeric vector that goes sequentially through the Embedding, LSTM, Dropout, and two Dense layers, suffering successive transformations and producing the final output. The LSTM+Dropout block of layers can be stacked n times to increase the learning capacity of the model.

Since each layer transforms its input differently, the contribution they bring to the final result is also different. The Embedding layer is responsible for the creation of the embedding vectors for each element of the vocabulary. We chose the LSTM layer to learn the relationship between the tokens and their ordering, i.e., the order in which the tokens appear in the piece of code that is being processed. To diminish overfitting, we introduced the Dropout Layer after each LSTM Layer. Finally, the Dense Layers appear, respectively, to learn to classify the examples, i.e., the relationship between the sample and its label, and to transform the vector into a single value between 0 and 1, which corresponds to how probable is the example of being vulnerable to SQLi.

Instead of an LSTM layer, we could have chosen other layers that can learn sequential relationships, such as the BiRNN, or opt for another type of network based on attention, as many recent NLP DL models. However, we believe that, due to the additional complexity they bring, it is preferable to first experiment with more straightforward configurations known to work in similar tasks before creating an overly sophisticated network that might not improve the results.

The optimization function most suitable to a binary classification problem is the Binary Cross Entropy Loss function [12]. Equation 3.1 represents this loss function, where y is the label vector (1 for vulnerable and 0 for non-vulnerable), $p(y_i)$ is the predicted probability of the example being vulnerable, and N is the number of examples.

$$H_p(y) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log_2(p(y_i)) + (1-y_i)\log_2(1-p(y_i))) \tag{3.1}$$

Thus, the network should have at least three layers, with Binary Cross Entropy Loss function.

Table 3.2: Definition of each layer's input, output and activation function.

Layer	Input	Output	Activation
Embedding	MAX_LENGTH, 1	MAX_LENGTH, HIDDEN_SIZE$_1$	-
LSTM$_1$	MAX_LENGTH, HIDDEN_SIZE$_1$	HIDDEN_SIZE$_1$, 1	Tanh
LSTM$_i$	HIDDEN_SIZE$_{i-1}$	HIDDEN_SIZE$_i$, 1	Tanh
Dropout$_i$	HIDDEN_SIZE$_i$, 1	HIDDEN_SIZE$_i$, 1	-
Dense	HIDDEN_SIZE$_n$, 1	HIDDEN_SIZE$_n$, 1	ReLU
Dense	HIDDEN_SIZE, 1	1, 1	Sigmoid

Table 3.2 lists each layer's input and output sizes, and activation function. The first LSTM layer receives as input a matrix (of size *MAX_LENGTH×HIDDEN_SIZE*) and outputs a vector (of size *HIDDEN_SIZE×1*). *MAX_LENGTH* corresponds to the maximum sequence size we allow and it is fixed before training the model to the longest sequence in the training set. *HIDDEN_SIZE*$_i$ is a predefined value that needs to be tuned along with n, where $i \in \{1, \dots, n\}$, and it represents the number of neurons in layer i. The subsequent LSTM and Dropout layers simply transform their input into vectors of the same size. The choice of activation function of each layer follows the recommendations of the Keras documentation [12] for the LSTM layer and of Goodfellow et al. [20] for the Dense layers. They state that the *ReLU* is commonly the preferred activation function for neural network layers and *Sigmoid* for output layers in classification problems with two classes. Furthermore, we chose to optimize the network with the Binary Cross Entropy Loss function, as explained in the previous section.

Next, we present each layer in detail.

Embedding Layer

The Embedding layer is responsible for mapping the tokens that constitute the vocabulary to embedding vectors. This layer receives a matrix and can simply function as a lookup table to match the token to the embedding vector (*static* approach), or it can change the embedding vectors during training (*dynamic* approach). The static approach is only feasible when we have the corresponding embedding vectors, like when pre-training them through the Word2Vec model. Since we could not build a big enough dataset to pre-train the embeddings, we chose the *dynamic* approach. The matrix is firstly initialized from a uniform distribution [12] and its values are updated according to the backpropagation algorithm and the training data [11].

LSTM Layer

The LSTM layer follows the configuration presented in Section 2.3, where it processes the tokens sequentially and learns patterns related to their order. In the vulnerability detection task, the order of the tokens in the sample is sometimes very relevant. For instance, it matters if the input is sanitized before being used in a sensitive sink. Furthermore, since we cannot predict how long these dependencies are, it is crucial to choose a layer that can deal with long-range dependencies, like the LSTM layer. Therefore, after mapping each token to the embedding vectors with the Embedding layer, the model will encode patterns present in the sample related to the order of the tokens with the LSTM layer.

The network presents n LSTM layers, where $n \in \mathbb{N}$. This flexibility in the network allows us to follow Chollet's recommendations on how to balance underfitting and overfitting in neural networks [11]. The author suggests that users begin with a single layer and then add layers until the performance starts to degrade. We can think of it as each layer is responsible for learning more details about the sample, where shallower layers learn coarser patterns, and deeper layers learn thinner patterns. We need to balance between learning in a coarse enough manner to be representative (avoid overfitting) while being specific enough to label the example correctly (avoid underfitting).

Dropout Layer

The Dropout layer is introduced after each LSTM layer to reduce overfitting. During training, the layer randomly sets some entries of its input to zero, according to a given probability (δ). This approach introduces noise in the model, preventing it from memorizing irrelevant patterns that are too sample-specific, learned by the previous layers. When evaluating an instance, the layer does not apply any transformation to its input [11], working as a simple identity layer, where the vector is transformed in itself.

Although it takes more time for the model to converge, neural network models that have dropout layers can reduce overfitting further and improve their performance [46].

Dense Layers

The last two layers are fully-connected feed-forward neural network layers, i.e., their neurons are all connected to all neurons from previous and next layers. These are core elements of neural networks, where they apply a simple transformation of the form

$$output = activation(dot(input, weights) + bias).$$

weights and *bias* are, respectively, a matrix and a vector created by the layer which start to be randomly initialized from a uniform distribution and are updated during training according to the backpropagation algorithm. The *input* and *output* vectors may have the same or different sizes. In our case, as we saw back in Table 3.2, the first Dense layer preserves the sizes while the second transforms the vector in a scalar.

The first Dense layer is introduced to learn the relationship between the example and its label (vulnerable or non-vulnerable). It transforms the input through the *ReLU* activation function in a vector with the same shape, which encodes this relationship. The last layer classifies the example by transforming the input in a value between 0 and 1. The value corresponds to how likely it is of being vulnerable to SLQi, and it is computed by the *Sigmoid* activation function, which is commonly used in binary classification tasks as ours to produce the final output [11].

To sum up, the model we created, although it may seem simple, is based on previous literature where it worked for similar tasks. It is a sequential DL model, with a minimum of five layers, which can grasp patterns in the order of the components of the input (thorugh the LSTM layers) and the relationship of the sample with its label (through the Dense layers).

3.3 Datasets

In this section, we present the datasets built by us. This process was done incrementally starting in a simpler representation featuring only the opcodes, to a representation that resulted from a more complex processing, where we have sequences of opcodes and operands with linear control flow (which we call *slices*). The dataset construction was one of the main focus and contribution of this thesis.

Thus, we start by extracting PHP code excerpts from SARD to create the first dataset - the PHP Excerpt Dataset (PED). We then used a tool to obtain the bytecode for each excerpt, which resulted in the Bytecode Excerpt Dataset (BED). The first dataset used to train the network was the Opcode Dataset (OD). As explained before, this dataset's samples are simply composed of sequences of opcodes from the BED. Although it is a simplistic representation, we consider it was an important starting point to assert some information about the behaviour of the network, such as which optimization algorithm to use. Following the experiments with the OD, and tending to our knowledge on SQLi, we decided to: 1) incorporate the operands, and built the Opcode + Operand Dataset (OOD), and 2) create a dataset where the bytecode statements are executed sequentially, that is, where the examples have linear control flow. To that matter, we built the Slice Dataset (SD). Finally, we created the Simplified Slice Dataset (SSD) where we restrict the opcode space by grouping opcodes we consider that have the same meaning when detecting SQLi.

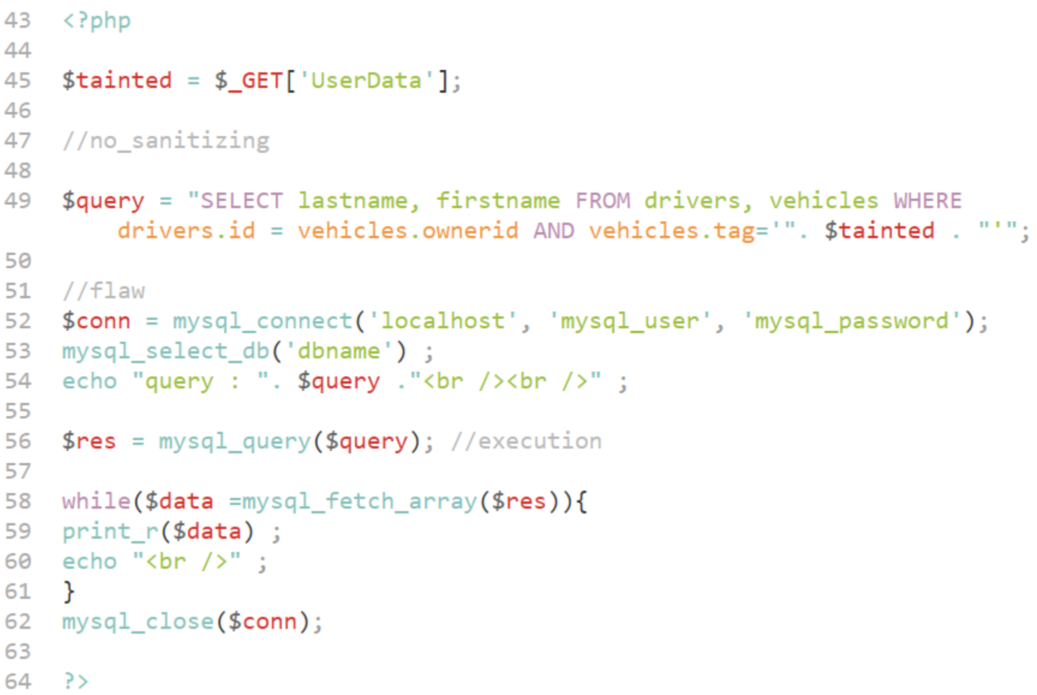
Summing up, we constructed six datasets along an incremental process, which four of them will be assessed with our model in order to determine the one that leads to a higher accuracy. These datasets are listed below and detailed in next sections:

- PHP Excerpt Dataset (PED)
- Bytecode Excerpt Dataset (BED)
- Opcode Dataset (OD)
- Opcode + Operand Dataset (OOD)
- Slice Dataset (SD)
- Simplified Slice Dataset (SSD)

3.3.1 PHP Excerpt Dataset (PED)

Stivalet and Fong [47] developed a PHP test case generation tool and made it available in SARD. Each sample is composed of a comment section labeling, a description and a code excerpt. The code excerpt starts with an entry point and ends with a sensitive sink.

In our work, we use SQLi test cases from SARD to compose the PHP Excerpt Dataset (PED). There are a total of 1362 instances - 858 vulnerable and 504 non-vulnerable. Non-vulnerable instances are code excerpts where the user input is correctly sanitized or validated (e.g., through the PHP function *mysqli_real_escape_string*). On the other hand, vulnerable excerpts lack input sanitization or validation, or such operations do not avoid malicious inputs effectively. Note that 1) the input may be



```php
<?php

$tainted = $_GET['UserData'];

//no_sanitizing

$query = "SELECT lastname, firstname FROM drivers, vehicles WHERE drivers.id = vehicles.ownerid AND vehicles.tag='". $tainted . "'";

//flaw
$conn = mysql_connect('localhost', 'mysql_user', 'mysql_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);

?>
```

Figure 3.11: Example of a PED instance, without the comment section.

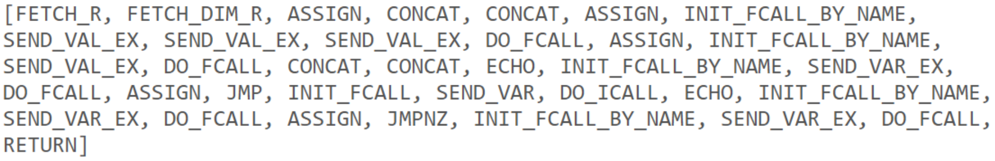
sanitized and still compromise the application, and 2) in vulnerable examples, the malicious input may be propagated across the excerpt through assignments to other variables. These show the complexity of the task we intend to solve.

Figure 3.11 shows an instance of the PED, without the comment section, for simplification purposes. This excerpt is labeled as vulnerable. It is easy to understand why it is effectively vulnerable considering that the variable `$tainted` receives external input through the global variable `$_GET['UserData']` (entry point) and it is used in the query statement without any type of validation or sanitization (stored in `$query`). This query is thereafter sent to the database by the *mysql_query* function (sensitive sink) to be executed there, exposing the application to a SQLi vulnerability. An attacker can exploit this vulnerability by providing, for example, `' OR 1=1'` as input, which will retrieve all drivers' first and last names from the database.

3.3.2 Bytecode Excerpt Dataset (BED)

We started by executing all examples from the PED on VLD, obtaining a bytecode excerpt for each code excerpt. These bytecode excerpts constitute the Bytecode Excerpt Dataset (BED). We will use this dataset to obtain the datasets described in the following sections, and that will be used to train the model.

Figure 2.3 shows the bytecode excerpt obtained on VLD for the PED example in Figure 3.11. In Section 2.2, we explained the different components of this representation. As stated in that section, the use of an IL simplifies the analysis by the compiler. Similarly, we believe it could also simplify the analysis that needs to be performed by the model. Besides, this approach can surely be applied to any high-level language that can be represented by the same IL.



```
[FETCH_R, FETCH_DIM_R, ASSIGN, CONCAT, CONCAT, ASSIGN, INIT_FCALL_BY_NAME,
SEND_VAL_EX, SEND_VAL_EX, SEND_VAL_EX, DO_FCALL, ASSIGN, INIT_FCALL_BY_NAME,
SEND_VAL_EX, DO_FCALL, CONCAT, CONCAT, ECHO, INIT_FCALL_BY_NAME, SEND_VAR_EX,
DO_FCALL, ASSIGN, JMP, INIT_FCALL, SEND_VAR, DO_ICALL, ECHO, INIT_FCALL_BY_NAME,
SEND_VAR_EX, DO_FCALL, ASSIGN, JMPNZ, INIT_FCALL_BY_NAME, SEND_VAR_EX, DO_FCALL,
RETURN]
```

Figure 3.12: OD instance, obtained from the BED example of Figure 2.3

3.3.3 Opcode Dataset (OD)

To analyze SLQi vulnerabilities, not all the information in a BED instance file is equally relevant. Therefore, our first approach was to create from each BED instance a sequence of its opcodes, i.e., a vector containing the opcodes from the *op* column (see Figure 2.3), maintaining the order in which they appear there, and with the same length as the *op* column. Hence, the Opcode Dataset (OD) is composed of the same number of vulnerable and non-vulnerable instances of PED and BED. Therefore, OD comprises 858 vulnerable and 504 non-vulnerable examples, where each one is a sequence of opcodes of variable length.

Figure 3.12 shows how a OD instance looks like, using the example we have been following (Figures 3.11 and 2.3). Table 3.3 lists all 198 opcodes from the OD, which will constitute the tokens of the vocabulary for our model (more details in Section 3.2.2). Basically, these are all the known opcodes from VLD to which we added an *out of vocabulary* token (`OOD`) in case of an unknown opcode appears during the evaluation of the model.

Table 3.3: Vocabulary for the Opcode Dataset.

OOV	NOP	ADD	SUB
MULT	DIV	MOD	SL
SR	CONCAT	BW_OR	BW_AND
BW_XOR	BW_NOT	BOOL_NOT	BOOL_XOR
IS_IDENTICAL	IS_NOT_IDENTICAL	IS_EQUAL	IS_NOT_EQUAL
IS_SMALLER	IS_SMALLER_OR_EQUAL	CAST	QM_ASSIGN
ASSIGN_ADD	ASSIGN_SUB	ASSIGN_MUL	ASSIGN_DIV
ASSIGN_MOD	ASSIGN_SL	ASSIGN_SR	ASSIGN_CONCAT
ASSIGN_BW_OR	ASSIGN_BW_AND	ASSIGN_BW_XOR	PRE_INC
PRE_DEC	POST_INC	POST_DEC	ASSIGN
ASSIGN_REF	ECHO	GENERATOR_CREATE	JMP
JMPZ	JMPNZ	JMPZNZ	JMPZ_EX
JMPNZ_EX	CASE	CHECK_VAR	SEND_VAR_NO_REF_EX
MAKE_REF	BOOL	FAST_CONCAT	ROPE_INIT
ROPE_ADD	ROPE_END	BEGIN_SILENCE	END_SILENCE
INIT_FCALL_BY_NAME	DO_FCALL	INIT_FCALL	RETURN
RECV	RECV_INIT	SEND_VAL	SEND_VAR_EX
SEND_REF	NEW	INIT_NS_FCALL_BY_NAME	FREE
INIT_ARRAY	ADD_ARRAY_ELEMENT	INCLUDE_OR_EVAL	UNSET_VAR
UNSET_DIM	UNSET_OBJ	FE_RESET_R	FE_FETCH_R
EXIT	FETCH_R	FETCH_DIM_R	FETCH_OBJ_R
FETCH_W	FETCH_DIM_W	FETCH_OBJ_W	FETCH_RW
FETCH_DIM_RW	FETCH_OBJ_RW	FETCH_IS	FETCH_DIM_IS
FETCH_OBJ_IS	FETCH_FUNC_ARG	FETCH_DIM_FUNC_ARG	FETCH_OBJ_FUNC_ARG
FETCH_UNSET	FETCH_DIM_UNSET	FETCH_OBJ_UNSET	FETCH_LIST
FETCH_CONSTANT	GOTO	EXIT_STMT	EXT_FCALL_BEGIN
EXT_FCALL_END	EXT_NOP	TICKS	SEND_VAR_NO_REF
CATCH	THROW	FETCH_CLASS	CLONE
RETURN_BY_REF	INIT_METHOD_CALL	INIT_STATIC_METHOD_CALL	ISSET_ISEMTY_VAR
ISSET_ISEMPTY_DIM_OBJ	SEND_VAL_EX	SEND_VAR	INIT_USER_CALL
UNKNOWN[119]	SEND_USER	STRLEN	DEFINED
TYPE_CHECK	VERIFY_RETURN_TYPE	FE_RESET_RW	FE_FETCH_RW
FE_FREE	INIT_DYNAMIC_CALL	DO_ICALL	DO_UCALL
DO_FCALL_BY_NAME	PRE_INC_OBJ	PRE_DEC_OBJ	POST_INC_OBJ
POST_DEC_OBJ	ASSIGN_OBJ	OP_DATA	INSTANCEOF
DECLARE_CLASS	DECLARE_INHERITED_CLASS	DECLARE_FUNCTION	RAISE_ABSTRACT_ERROR
DECLARE_CONST	ADD_INTERFACE	VERIFY_INSTANCEOF	VERIFY_ABSTRACT_CLASS
ASSIGN_DIM	ISSET_ISEMPTY_PROP_OBJ	HADLE_EXCEPTION	USER_OPCODE
ASSERT_CHECK	JMP_SET	DECLARE_LAMBDA_FUNCTION	ADD_TRAIT
BINDTRAIS	SEPARATE	FETCH_CLASS_NAME	JMP_SET_VAR
DISCARD_EXCEPTION	YIELD	GENERATOR_RETURN	FAST_CALL
FAST_RET	RECV_VARIADIC	SEND_UNPACK	POW
ASSIGN_POW	BIND_GLOBAL	COALESCE	SPACESHIP
DECLARE_ANON_CLASS	DECLARE_ANON_INHERITED_CLASS	FETCH_STATIC_PROP_R	FETCH_STATIC_PROP_W
FETCH_STATIC_PROP_RW	FETCH_STATIC_PROP_IS	FETCH_STATIC_PROP_FUC_ARG	FETCH_STATIC_PROP_UNSET
UNSET_STATIC_PROP	ISSET_ISEMPTY_STATIC_PROP	FETCH_CLASSICAL_CONSTANT	BIND_LEXICAL
BIND_STATIC	FETCH_THIS	UNKNOWN[185]	ISSET_ISEMPTY_THIS
SWITCH_LONG	SWITCH_STRING	IN_ARRAY	COUNT
GET_CLASS	GET_CALLED_CLASS	GET_TYPE	FUNC_NUM_ARGS
FUNC_GET_ARGS	ISSET_EMPTY		

3.3.4 Upgrades to the Opcode Dataset

After experimenting with the OD, we detected two potential problems:

1. *Absence of operands:* Analyzing the opcodes without their operands may be insufficient to detect SQLi vulnerabilities - we cannot extract data flow nor function names (these only appear in the operands of certain opcodes like `INIT_FCALL`);

2. *Control flow:* Although the opcodes from an instance of the OD are processed sequentially by the model, they may not be executed sequentially, as in the case of `if` and `while` statements.

To overcome these two issues, we propose 1) adding the opcodes' operands, and 2) convert excerpts into slices, where a slice represents an execution path of the excerpt. Each solution originated a new dataset, respectively, the Opcode + Operand Dataset (OOD), and the Slice Dataset (SD).

Additionally to these two datasets, we built a third that results from the SD by group-ing some opcodes that we believe that have similar meaning in the context of SQLi detection. We named it Simplified Slice Dataset (SSD). Let us look at these three datasets in more detail.

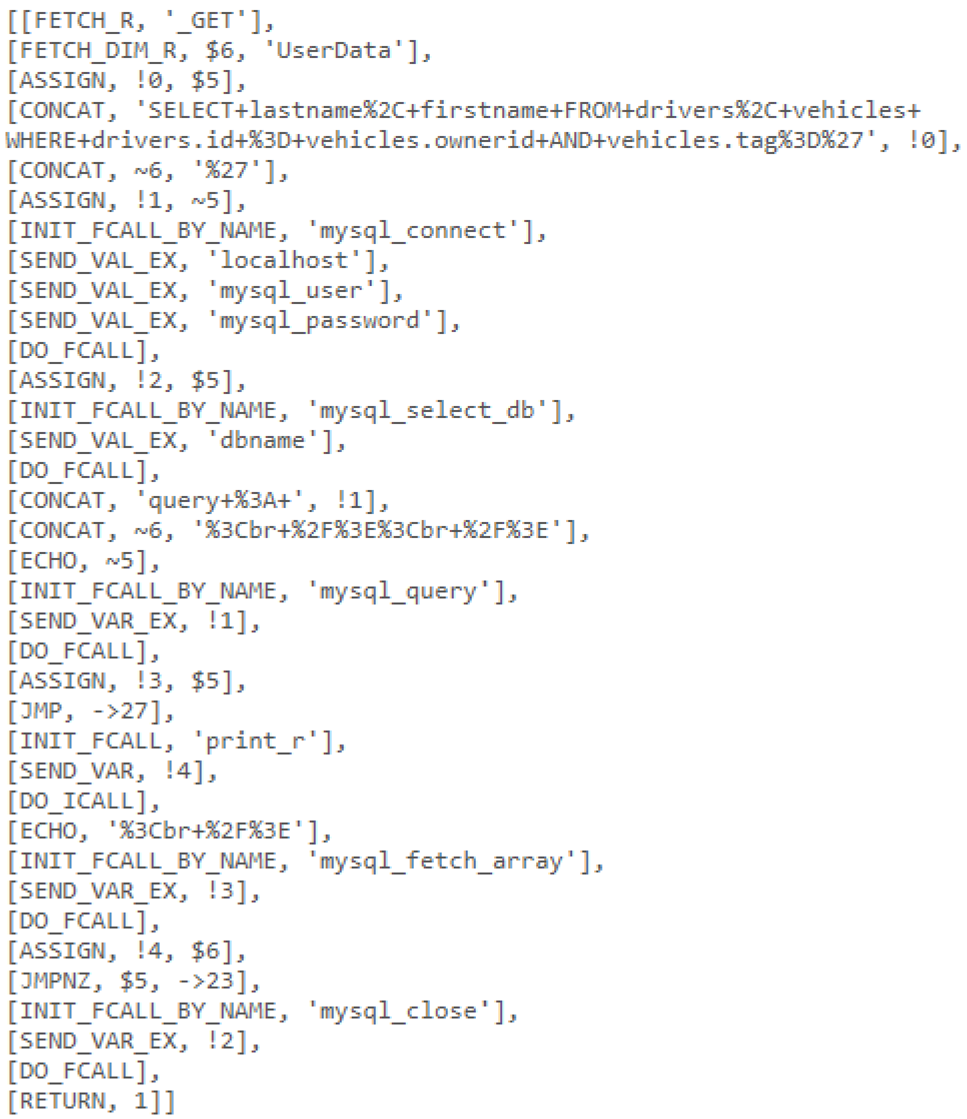
1. *Opcode + Operand Dataset (OOD)*

We constructed the OOD dataset to solve the absence of operand, which is pretty straight forward. Each example is composed of a sequence of bytecode statements, where a statement is itself a sequence of one or more tokens. The first token is always the opcode, followed by its operands if it has any. Again, the examples may have different lengths, and their statements too. To construct the dataset, each BED sample suffers a two step process. First, from each bytecode statement it is extracted the opcode and its operands. Then, each operand is analyzed and may be translated to a grouped operand token, as explained further down in this section. Each instance is represented by a vector in which each element corresponds to a bytecode statement, where it is also expressed by a vector.

Figure 3.13 depicts the result of the first step when applied to the sequence in Figure 2.3, where we can observe that the resulting vector comprises a set of subvectors to represent the bytecode statements.

Analysis of the Operands

Note that the operands' space is potentially infinite. One must only think of the amount possibilities there are for strings and numeric values in PHP. This fact leads



```
[[FETCH_R, '_GET'],
[FETCH_DIM_R, $6, 'UserData'],
[ASSIGN, !0, $5],
[CONCAT, 'SELECT+lastname%2C+firstname+FROM+drivers%2C+vehicles+
WHERE+drivers.id+%3D+vehicles.ownerid+AND+vehicles.tag%3D%27', !0],
[CONCAT, ~6, '%27'],
[ASSIGN, !1, ~5],
[INIT_FCALL_BY_NAME, 'mysql_connect'],
[SEND_VAL_EX, 'localhost'],
[SEND_VAL_EX, 'mysql_user'],
[SEND_VAL_EX, 'mysql_password'],
[DO_FCALL],
[ASSIGN, !2, $5],
[INIT_FCALL_BY_NAME, 'mysql_select_db'],
[SEND_VAL_EX, 'dbname'],
[DO_FCALL],
[CONCAT, 'query+%3A+', !1],
[CONCAT, ~6, '%3Cbr+%2F%3E%3Cbr+%2F%3E'],
[ECHO, ~5],
[INIT_FCALL_BY_NAME, 'mysql_query'],
[SEND_VAR_EX, !1],
[DO_FCALL],
[ASSIGN, !3, $5],
[JMP, ->27],
[INIT_FCALL, 'print_r'],
[SEND_VAR, !4],
[DO_ICALL],
[ECHO, '%3Cbr+%2F%3E'],
[INIT_FCALL_BY_NAME, 'mysql_fetch_array'],
[SEND_VAR_EX, !3],
[DO_FCALL],
[ASSIGN, !4, $6],
[JMPNZ, $5, ->23],
[INIT_FCALL_BY_NAME, 'mysql_close'],
[SEND_VAR_EX, !2],
[DO_FCALL],
[RETURN, 1]]
```

Figure 3.13: Sequence of opcodes and operands obtained from the BED example depicted in Figure 2.3

to a problem: the vocabulary for this dataset will be huge, and filled with tokens that appear only once and will possibly never appear again. A model with a vocabulary like this will struggle to perform well. Also, certain differences do not affect the classification of the instances. Therefore, we created 6 tokens that can represent the majority of the operand tokens, and which we believe do not lose too much information from the sequence.

Table 3.4 presents the grouped operand tokens we defined, resulting from the analysis we made over the operands of our dataset. We see that, for instance, the token `<FNC>` represents any function except those dedicated to SQL and filtering. For these two cases, we defined two extra tokens: `SQL_FNC` and `FTR_FNC`.

Another important aspect of these transformations is that it keeps some operand

Table 3.4: List of grouped operands and corresponding token.

Token	**Operand**
<FNC>	Corresponds to a function name Does not contain the string 'sql' or is equal to 'filter_input', 'filter_var_array' or ' filter_var'
<SQL_FNC>	Corresponds to a function name Contains the string 'sql'
<FTR_FNC>	Corresponds to 'filter_input', 'filter_var_array' or ' filter_var'
<QRY>	Corresponds to a SQL query
<NUM>	A numerical value that is not part of a <FTR_FNC>statements
<STR>	Any operand that does not fall in any other category Cannot be a variable (variables follow the pattern `'(^[!~\$]|^->|^:-)\d+\$'`)

tokens unaltered, namely variable tokens (since keeping track of the data flow is crucial to our task) and numeric values that represent PHP constants for filtering. In our dataset there are only four: 274, 515, 517, and 522, which correspond to `FILTER_VALIDATE_EMAIL`, `FILTER_SANITIZE_SPECIAL_CHARS`, `FILTER_SANITIZE_EMAIL`, and `FILTER_SANITIZE_FULL_SPECIAL_CHARS`, respectively.

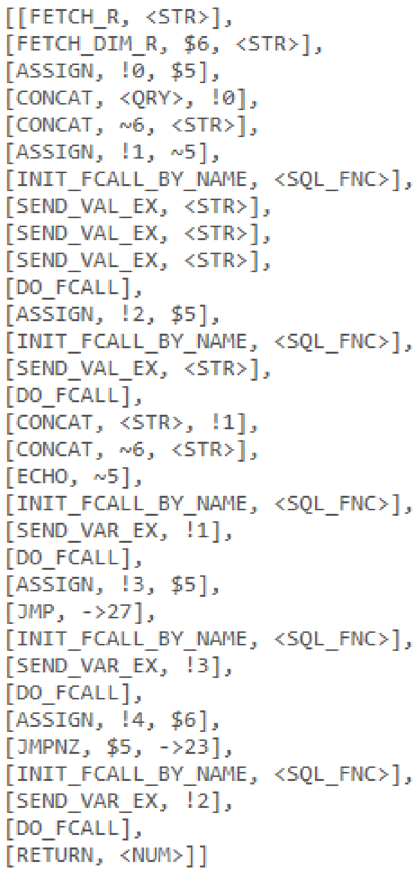
Figure 3.14 represents the OOD instance for the BED example in Figure 2.3. Each line corresponds to a statement, which is a sequence of tokens. It is easy to confirm that the first token is always the opcode of the corresponding statement. Note that the statements have variable length, as mentioned above. The first statement is composed of two tokens - the opcode and a single operand (where the *_GET* operand is translated to the `<STR>` token), while the second one has length three - the opcode and two operands.

The final dataset's *base* vocabulary contains 208 tokens (198 opcode tokens presented in Table 3.3, 6 operand tokens defined in Table 3.4, and 4 tokens corresponding to PHP filter constants). Apart from these tokens, others may be added before training, corresponding to variable tokens that appear in the train dataset, and that are left unaltered.

2. *Slice Dataset (SD)*

A slice represents an execution path, and besides granting linear control flow, which reduces the representation complexity, its structure is now closer to natural language, allowing better performance with NLP models such as the ones we will use. This strategy can, in a simple way, solve the second issue presented at the beginning of this subsection - the control flow problem.

Since the same excerpt may have several execution paths, a bytecode excerpt may also originate several slices. To obtain the slice dataset, we manually processed each



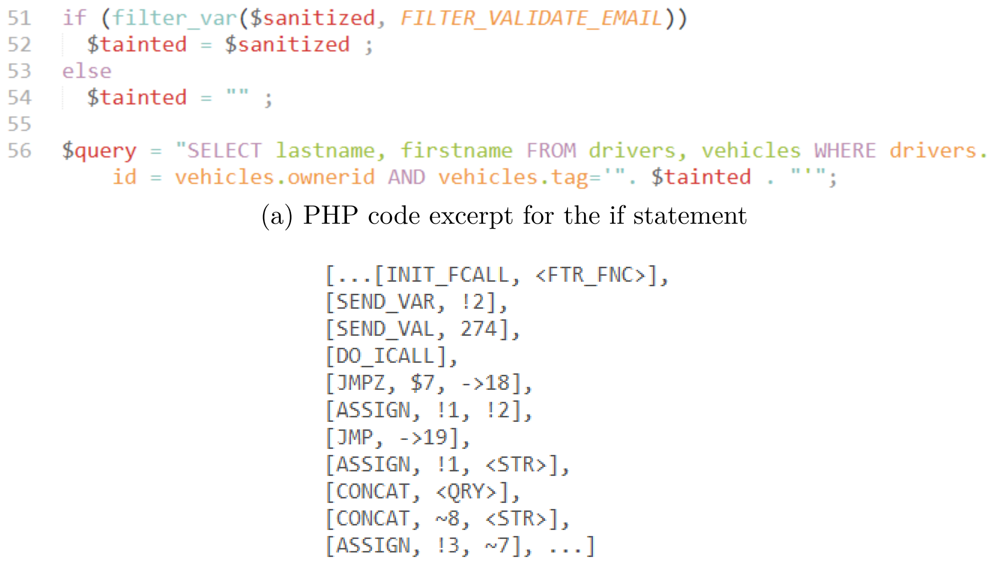
```
[[FETCH_R, <STR>],
[FETCH_DIM_R, $6, <STR>],
[ASSIGN, !0, $5],
[CONCAT, <QRY>, !0],
[CONCAT, ~6, <STR>],
[ASSIGN, !1, ~5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAL_EX, <STR>],
[SEND_VAL_EX, <STR>],
[SEND_VAL_EX, <STR>],
[DO_FCALL],
[ASSIGN, !2, $5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAL_EX, <STR>],
[DO_FCALL],
[CONCAT, <STR>, !1],
[CONCAT, ~6, <STR>],
[ECHO, ~5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !1],
[DO_FCALL],
[ASSIGN, !3, $5],
[JMP, ->27],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !3],
[DO_FCALL],
[ASSIGN, !4, $6],
[JMPNZ, $5, ->23],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !2],
[DO_FCALL],
[RETURN, <NUM>]]
```

Figure 3.14: OOD instance. Obtained from the example depicted in 2.3 after applying the transformations in Table 3.4

bytecode excerpt. We analyzed 522 excerpts from which we obtained 1650 slices, 772 of which are vulnerable. Afterwards, we obtained the sequence of opcodes and operands following the same transformations as for the OOD.

It is important to note that, even though the excerpts are already classified, the final slice classification may be different. An excerpt without SQLi vulnerabilities always originates slices free of vulnerabilities. However, a vulnerable excerpt may have one or more slices that do not create SQLi vulnerabilities. For instance, the variable containing the vulnerable input may be sanitized inside an if statement, leading to a vulnerable slice (without the branch with the sanitization) and a non-vulnerable slice (with the branch with the sanitization). Hence, the classification of each slice was also manually done.

To produce the excerpts' slices, we need to look for control flow statements such as `if` and `while`. In more complex applications, other statements would be of interest, such as `switch cases` and exceptions, but in our dataset there are only `if` and `while` statements since SARD test cases are small and simple. Therefore, we will only detail these two. However, the process we present next can be extended and applied to other control flow statements.



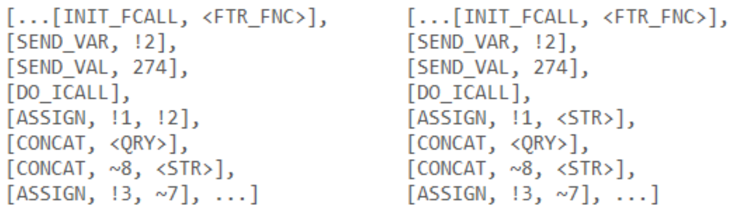
```
51  if (filter_var($sanitized, FILTER_VALIDATE_EMAIL))
52    $tainted = $sanitized ;
53  else
54    $tainted = "" ;
55
56  $query = "SELECT lastname, firstname FROM drivers, vehicles WHERE drivers.
        id = vehicles.ownerid AND vehicles.tag='". $tainted . "'";
```

(a) PHP code excerpt for the if statement

```
[...[INIT_FCALL, <FTR_FNC>],
[SEND_VAR, !2],
[SEND_VAL, 274],
[DO_ICALL],
[JMPZ, $7, ->18],
[ASSIGN, !1, !2],
[JMP, ->19],
[ASSIGN, !1, <STR>],
[CONCAT, <QRY>],
[CONCAT, ~8, <STR>],
[ASSIGN, !3, ~7], ...]
```

(b) OOD excerpt for the if statement

Figure 3.15: Example of an if statement in PHP code and opcodes + operands.



```
[...[INIT_FCALL, <FTR_FNC>],
[SEND_VAR, !2],
[SEND_VAL, 274],
[DO_ICALL],
[ASSIGN, !1, !2],
[CONCAT, <QRY>],
[CONCAT, ~8, <STR>],
[ASSIGN, !3, ~7], ...]
```

```
[...[INIT_FCALL, <FTR_FNC>],
[SEND_VAR, !2],
[SEND_VAL, 274],
[DO_ICALL],
[ASSIGN, !1, <STR>],
[CONCAT, <QRY>],
[CONCAT, ~8, <STR>],
[ASSIGN, !3, ~7], ...]
```

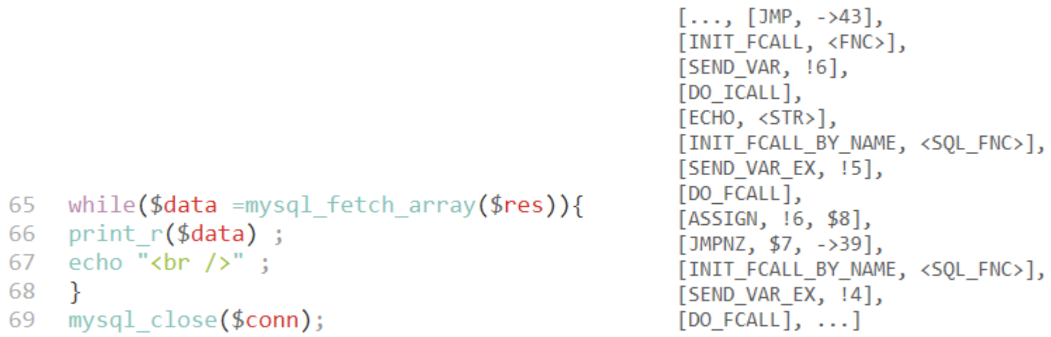
(a) Slice corresponding to the value *true*.

(b) Slice corresponding to the value *false*.

Figure 3.16: Example of an if statement's slices.

Addressing `if` statements

The presence of an `if` statement originates two slices: one corresponding to the value true of the if condition and another one corresponding to the value false. Figure 3.15 shows how an if statement is processed and separated into two slices. In Figure 3.15 a), there are a few lines taken from a sample of the PED. Figure 3.15 b) shows the corresponding sequence of opcodes and operands. Figure 3.16 shows the two slices extracted where: 1) Figure 3.16 a) includes only the statement for code line 52 (the instruction of the true branch), and 2) Figure 3.16 b) contains the statement for code line 54 (the instruction of the false branch). On the other hand, since the condition is executed whether it is true or false, both a) and b) contain the first fours statements, which correspond to that line (line 51).



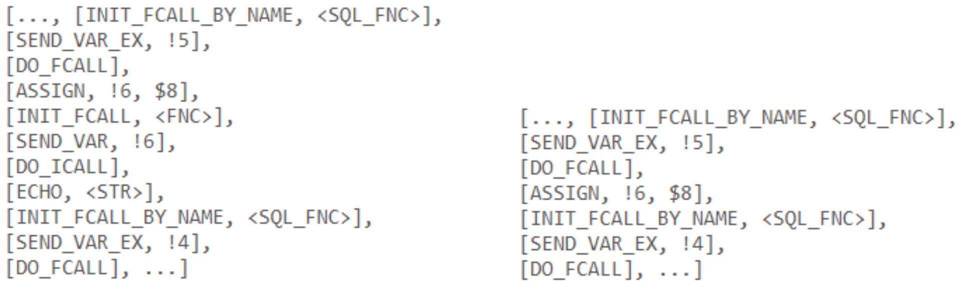
```
65  while($data =mysql_fetch_array($res)){
66  print_r($data) ;
67  echo "<br />" ;
68  }
69  mysql_close($conn);
```

(a) PHP code excerpt for the while statement

```
[..., [JMP, ->43],
[INIT_FCALL, <FNC>],
[SEND_VAR, !6],
[DO_ICALL],
[ECHO, <STR>],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !5],
[DO_FCALL],
[ASSIGN, !6, $8],
[JMPNZ, $7, ->39],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !4],
[DO_FCALL], ...]
```

(b) OOD excerpt for the while statement

Figure 3.17: Example of a while statement PHP code and opcodes + operands.



```
[..., [INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !5],
[DO_FCALL],
[ASSIGN, !6, $8],
[INIT_FCALL, <FNC>],
[SEND_VAR, !6],
[DO_ICALL],
[ECHO, <STR>],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !4],
[DO_FCALL], ...]
```

(a) Slice corresponding to the value *true*.

```
[..., [INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !5],
[DO_FCALL],
[ASSIGN, !6, $8],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !4],
[DO_FCALL], ...]
```

(b) Slice corresponding to the value *false*.

Figure 3.18: Example of a while statement's slices.

Addressing `while` statements

To deal with `while` statements, we also decided to produce two slices from the original excerpt: the first corresponds to when the while condition is false - therefore, the while block is not executed - and the second corresponds to the case where the while block is executed once. We can proceed this way because, to detect a vulnerability, a single representation of the block suffices. Figure 3.17 shows a while statement taken from a PED example and the corresponding opcode + operand sequence. Figure 3.18 shows how this while statement is divided into two slices. Note that in Figure 3.17 b) the while condition (code line 65) appears after the while block. However, to obtain a linear control flow, these lines precede the while block in the slices (first four statements in Figures 3.18 a) and b)).

Note that, because the slices have linear control flow, all opcodes related to control flow changes, such as jumps, are irrelevant and were excluded from the vocabulary. That is why in Figure 3.16 the `JMP` and `JMPZ` disappeared, and in Figure 3.18 the `JMP` and `JMPNZ` disappeared as well. Listing 3.1 shows all excluded opcodes. This means that the SD base vocabulary has $208 - 11 = 197$ tokens (the same tokens as OOD minus the control flow tokens). Again, additional tokens may be added before training, corresponding to variable tokens present in the training set.

```
JMP
JMPZ
JMPNZ
JMPZ_EX
JMPNZ_EX
CASE
JMP_SET
JMP_SET_VAR
GOTO
SWITCH_LONG
SWITCH_STRING
```

Listing 3.1: Opcodes excluded from the list of possible opcodes.

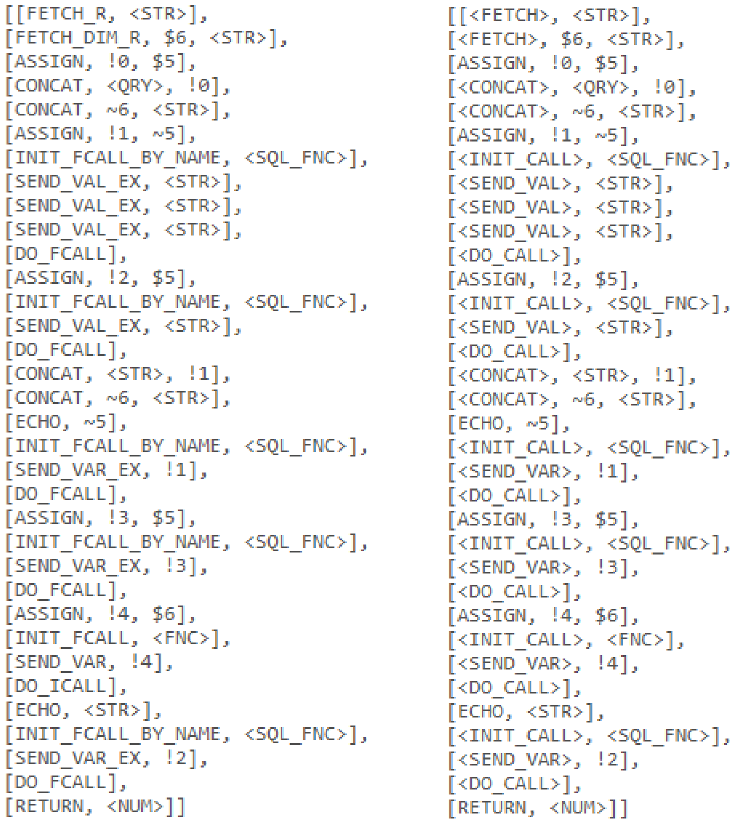
Some slices may have calls to user-defined functions. Therefore, it was necessary to include statement blocks from user functions, defined in or out of classes, to the correct sequence location where they will be executed.

3. ***Simplified Slice Dataset (SSD)***

We created a last dataset called Simplified Slice Dataset (SSD) that tries to prevent overfitting that may be caused by a large number of unique opcodes. Similarly to the procedure followed when creating the OOD samples, we grouped some opcodes that have a similar meaning to our task by representing them by the same token. For instance, there are numerous opcodes for mathematical operations, such as `ADD` and `MULT`, that correspond to the addition and multiplication, respectively. Nevertheless, to detect an SQLi vulnerability, it is normally irrelevant which one appears in the example. So, we represent such operations by the token <`OPER`>. Table 3.5 shows the tokens we defined to aggregate opcodes with similar functionalities. We defined 10 new tokens to represent 101 opcode tokens. Therefore, the SSD has $197 - 101 + 10 = 106$ unique tokens that form its base vocabulary, which correspond to SD base vocabulary tokens minus the old opcode tokens plus the new opcode tokens.

Figure 3.19 exemplifies the changes applied to an SD sample in the SSD. The figure shows how the example from Figure 2.3 is represented in the SD (Figure 3.19(a)) and SSD (Figure 3.19(b)) when the while statement in line 48 is true. Note that, for instance, the opcode in the first line of (a) is `FETCH_R`, which corresponds to the token <`FETCH`>, as shown in Table 3.5. The remaining tokens in that line remain equal, since they are operand tokens.

To sum up, we created six datasets: the PHP Excerpt Dataset, the Bytecode Excerpt Dataset, the Opcode Dataset, Opcode + Operand Dataset, the Slice Dataset,



```
[[FETCH_R, <STR>],
[FETCH_DIM_R, $6, <STR>],
[ASSIGN, !0, $5],
[CONCAT, <QRY>, !0],
[CONCAT, ~6, <STR>],
[ASSIGN, !1, ~5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAL_EX, <STR>],
[SEND_VAL_EX, <STR>],
[SEND_VAL_EX, <STR>],
[DO_FCALL],
[ASSIGN, !2, $5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAL_EX, <STR>],
[DO_FCALL],
[CONCAT, <STR>, !1],
[CONCAT, ~6, <STR>],
[ECHO, ~5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !1],
[DO_FCALL],
[ASSIGN, !3, $5],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !3],
[DO_FCALL],
[ASSIGN, !4, $6],
[INIT_FCALL, <FNC>],
[SEND_VAR, !4],
[DO_ICALL],
[ECHO, <STR>],
[INIT_FCALL_BY_NAME, <SQL_FNC>],
[SEND_VAR_EX, !2],
[DO_FCALL],
[RETURN, <NUM>]]
```

(a) SD example

```
[[<FETCH>, <STR>],
[<FETCH>, $6, <STR>],
[ASSIGN, !0, $5],
[<CONCAT>, <QRY>, !0],
[<CONCAT>, ~6, <STR>],
[ASSIGN, !1, ~5],
[<INIT_CALL>, <SQL_FNC>],
[<SEND_VAL>, <STR>],
[<SEND_VAL>, <STR>],
[<SEND_VAL>, <STR>],
[<DO_CALL>],
[ASSIGN, !2, $5],
[<INIT_CALL>, <SQL_FNC>],
[<SEND_VAL>, <STR>],
[<DO_CALL>],
[<CONCAT>, <STR>, !1],
[<CONCAT>, ~6, <STR>],
[ECHO, ~5],
[<INIT_CALL>, <SQL_FNC>],
[<SEND_VAR>, !1],
[<DO_CALL>],
[ASSIGN, !3, $5],
[<INIT_CALL>, <SQL_FNC>],
[<SEND_VAR>, !3],
[<DO_CALL>],
[ASSIGN, !4, $6],
[<INIT_CALL>, <FNC>],
[<SEND_VAR>, !4],
[<DO_CALL>],
[ECHO, <STR>],
[<INIT_CALL>, <SQL_FNC>],
[<SEND_VAR>, !2],
[<DO_CALL>],
[RETURN, <NUM>]]
```

(b) SSD example

Figure 3.19: SD and SSD examples corresponding to the OOD example in Figure 3.14 when the while statement is true.

and the Simplified Slice Dataset. We used the first two datasets to understand how the SLQi detection could be carried out, and to build the remaining datasets. As for the rest, we used them to train and test the model. Figure 3.20 shows the relationship between different components we presented throughout this section. Continuous lines define dataset dependencies from other datasets. For example, the BED could not be originated without the PED. On the other hand, the BED is essential for the construction of the OD, the OOD, and the SD. Dashed lines indicate the datasets resort to an external resource, such as the SARD or the conversion in grouped operator tokens of Table 3.5.

Table 3.5: List of grouped opcodes and corresponding token.

Token	Opcode
<INIT_CALL>	INIT_FCALLINIT_FCALL_BY_NAME INIT_NS_FCALL_BY_NAME INIT_METHOD_CALL INIT_STATIC_METHOD_CALL INIT_USER_CALL INIT_DYNAMIC_CALL
<DO_CALL>	DO_FCALL DO_ICALL DO_UCALL DO_FCALL_BY_NAME
<DECLARE>	DECLARE_CLASS DECLARE_INHERITED_CLASS DECLARE_FUNCTION DECLARE_LAMBDA_FUNCTION DECLARE_ANON_CLASS DECLARE_ANON_INHERITED_CLASS DECLARE_CONST
<FETCH>	FECTH_R FECTH_DIM_R FECTH_OBJ_R FECTH_W FECTH_DIM_W FECTH_OBJ_W FECTH_RW FECTH_DIM_RW FECTH_OBJ_RW FECTH_IS FECTH_DIM_IS FECTH_OBJ_IS FECTH_FUNC_ARG FECTH_DIM_FUNC_ARG FECTH_OBJ_FUNC_ARG FECTH_UNSET FECTH_DIM_UNSET FECTH_OBJ_UNSET FECTH_LIST FECTH_CONSTANT FECTH_CLASS FECTH_CLASS_NAME FECTH_CLASSICAL_CONSTANT FECTH_STATIC_PROP_R FECTH_STATIC_PROP_W FECTH_STATIC_PROP_RW FECTH_STATIC_PROP_IS FECTH_STATIC_PROP_FUNC_ARG
<UNSET>	UNSET_STATIC_PROP UNSET_VARUNSET_DIM UNSET_OBJ
<OPER>	NOP ADD SUB MULT DIV MOD SL SR BW_OR BW_AND BW_XOR BW_NOT BOOL_XOR PRE_INC PRE_DEC POST_INC POST_DEC ASSIGN_ADD ASSIGN_SUB ASSIGN_MULT ASSIGN_DIV ASSIGN_MOD ASSIGN_SL ASSIGN_SR ASSIGN_BW_OR ASSIGN_BW_AND ASSIGN_BW_XOR POW ASSIGN_POW PRE_INC_OBJ PRE_DEC_OBJPOST_INC_OBJ POST_DEC_OBJ
<CONCAT>	CONCAT ASSIGN_CONCAT FAST_CONCAT
<COMPR>	IS_IDENTICAL IS_NOT_IDENTICAL IS_EQUAL IS_NOT_EQUAL IS_SMALLER IS_SMALLER_OR_EQUAL ISSET_ISEMPTY_PROP_OBJ ISSET_ISEMPTY_VAR ISSET_ISEMPTY_DIM_OBJ ISSET_ISEMPTY_STATIC_PROP ISSET_ISEMPTY_THIS ISSET_EMPTY
<SEND_VAR>	SEND_VAR SEND_VAR_NO_REF_EX SEND_VAR_EX SEND_VAR_NO_REF
<SEND_VAL>	SEND_VAL SEND_VAL_EX

In Table 3.6 we summarize the sizes of the datasets. The SD and SSD are the only ones that have different sizes. Although they were obtained from as little as 522 PHP excerpts, the dataset is larger and more balanced. We also present the size of the base vocabulary for each dataset. A base vocabulary comprises all previously

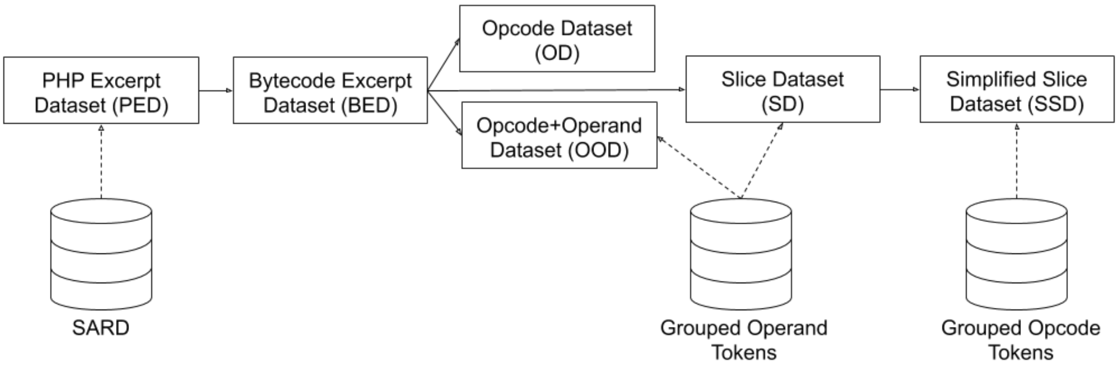

PHP Excerpt Dataset (PED)
Bytecode Excerpt Dataset (BED)
Opcode Dataset (OD)
Opcode+Operand Dataset (OOD)
Slice Dataset (SD)
Simplified Slice Dataset (SSD)
SARD
Grouped Operand Tokens
Grouped Opcode Tokens

Figure 3.20: Diagram representing the dependencies between the different datasets.

Table 3.6: Datasets sizes.

Dataset	**Vulnerable (%)**	**Total Size**	**Base Vocabulary Size**
PHP Excerpt Dataset	858 (63%)	1362	-
Bytecode Excerpt Dataset	858 (63%)	1362	-
Opcode Dataset	858 (63%)	1362	198
Opcode + Operand Dataset	858 (63%)	1362	*208
Slice Dataset	772 (47%)	1650	*197
Simplified Slice Dataset	772 (47%)	1650	*106

* the size is variable

known tokens of a given dataset, without looking at the training set. OD is the only dataset whose vocabulary is fixed. The rest, due to the different variable tokens that may appear in the train dataset, will have different sizes. Before training, we add to the vocabulary tokens that appear in the training set and not in the base vocabulary.

Chapter 4

Implementation

In this chapter, we will go over some implementation details. We will start with the construction of the datasets and then specify how the model configurations were built and evaluated. For a better understanding, we will use conceptual modelling and provide the algorithms implemented. Since the work was done in an iterative manner, starting with the OD and finishing with the SSD, some algorithms only represent the final implementation. We will also briefly explain how the algorithms evolved when necessary.

We used Python scripts to execute all tasks. Furthermore, we chose the well-known Python package Keras [12] to implement the various model configurations. Keras provides a convenient and easy-to-use interface for developing deep neural networks. It works as a wrapper for the Tensorflow package [9], which needs more details to configure a model.

4.1 Dataset Construction

In this section, we will take a closer look at the strategy used to build the datasets. After extracting the PHP code excerpts from SARD and obtaining the corresponding bytecode, we gathered them in two directories: one for the code excerpts (which comprise the PED), and another for the bytecode excerpts (the BED). The BED was then used to collect information to create the OD and OOD. Moreover, the BED was also utilized to create the BED with slices for the SD and SSD. For information extraction, we implemented *preprocess.py.* This script creates a dataset in a given directory by gathering the intended information from bytecode excerpts in the input directory (BED or BED with slices).

Algorithm 1 presents the necessary steps to create the datasets. This procedure receives as input the source directory where the bytecode files are located (*srcdir*), the destination directory where the dataset will be stored (*dstdir*), the information to include in the dataset (*info* - opcodes or opcodes+parameters), and two booleans

- one indicating whether we want to group the opcodes with similar meaning (*opcgroup*) and another indicating whether we want to group the operands belonging to the same category (*oprgroup*). For each bytecode file in the source directory, the algorithm extracts the bytecode (line 2) and then the intended information from that bytecode, producing a sequence of opcodes or a sequence of opcodes+operands (line 3). Afterwards, it applies the opcode and operand grouping taking into consideration the value of the variables *opcgroup* and *oprgroup* (lines 5 and 8). Finally, the resulting sequence is written to a file and stored in the *dstdir* directory.

```
   input  : A source directory with bytecode files srcdir;
            A destination directory dstdir for the new dataset;
            The information info to include in the new dataset;
            Booleans opcgroup/oprgroup indicating whether the
            opcodes/operands are grouped
   output: A set of files in directory dstdir
 1 foreach file in srcdir do
 2     code $\leftarrow$ GetCode(file);
 3     seq $\leftarrow$ GetSeq(code, info);
 4     if opcgroup then
 5         seq $\leftarrow$ opc2tkn.TransformOpcodes(seq)
 6     end
 7     if oprgroup then
 8         seq $\leftarrow$ opr2tkn.TransformOperands(seq)
 9     end
10     write seq to new file in dstdir;
11 end
```

Algorithm 1: How to create OD, OOD, SD, and SSD

To group opcodes and operands according to Tables 3.4 and 3.5, we created two Python modules - *opc2tkn.py* and *opr2tkn.py* - that group opcodes and operands to the corresponding tokens, respectively. The main functions of those modules are *TransformOpcodes* (module *opc2tkn.py*) and *TransformOperands* (module *opr2tkn.py*), which are executed in Algorithm 1 in lines 5 and 8. These two functions are similar: they receive the sequence of a bytecode file and, for each statement representation, they check whether it needs to be transformed or not. In the end, the functions return the sequence with the transformations. Other functions in the modules check which is the group of the opcode/operand being analyzed. These modules were used to group the operands when building the OOD and SD, and to group both the opcodes and operands for the SSD construction.

There are two functions that play a crucial role in this procedure: *GetCode and GetSeq*. Algorithms 2 and 3 specify how these functions were implemented. Function *GetCode* creates a dictionary containing the bytecode organized per class and function, i.e., the keys in the dictionary are class or function names, whereas the values are a list of statements if it is a function, or a dictionary with the functions of the corresponding class. Thus, we need to extract each function's definition from the bytecode file. A function definition starts with a line stating the class/function name. For example, the bytecode of a function named *Foo* would have a line, previous to the actual bytecode of the function, stating `Function Foo:`. Therefore, this function's bytecode would be stored as a sequence, in *dict*, under the key *Foo*. Analogously, classes' bytecodes are stored in *dict* under a key with their name. The difference is that, since classes normally have functions (or methods) themselves, their bytecode is stored in dictionaries instead of sequences, which means those dictionaries store the classes' functions. Algorithm 2 receives a bytecode file *file*. It starts by extracting the bytecode corresponding to the *main* part of the program, and add it to the dictionary *dict* under the key *main*. After the *main*, it is possible to have the user-defined functions and classes definitions. To store it in *dict*, we look for lines starting with the word *'Class'* or *'Function'*, as it indicates that the following bytecode belongs to that class/function. For those cases, the bytecode is stored in *dict* under a key with the name of the class/function.

This procedure is a little bit different for the OD. Since the OD was our first approach, we chose not to work with the user-defined functions' bytecode. Hence, we did not apply the *while* block of the algorithm.

```
input  : A bytecode file file
output: Dictionary dict with bytecode sequences for the bytecode in file file
$i \leftarrow 0$;
$i, main \leftarrow$ GetMainCode($i$);
dict[$'main'$] $\leftarrow main$;
$i \leftarrow i + 1$;
while $i <$ number of file lines do
    if first word of line i is 'Class' then
        $class\_name \leftarrow$ GetClassName($i$);
        dict [class_name] $\leftarrow$ GetClassCode();
    end
    if first word of line i is 'Function' then
        $function\_name \leftarrow$ GetFunctionName($i$);
        dict [function_name] $\leftarrow$ GetFunctionCode();
    end
    $i \leftarrow i + 1$;
end
return dict;
```

Algorithm 2: Get the bytecode from a bytecode file (*GetCode*)

Algorithm 3 gives the details for the function *GetSeq*, where the final sequence *seq* is constructed. This sequence is our representation of the bytecode for the corresponding excerpt. In this algorithm, we feed the function with the bytecode dictionary *dict* and the information *info* we want to extract. The sequence starts as an empty list. Next, we go through the statements under the key *main* stored in *dict*. Note that, as defined in Section 2.2, we call instruction to a line of code and statment to a line of bytecode, i.e., an instruction may be translated into multiple statements. If the statement corresponds to a call of a user-defined function (note that it may be defined inside a class), it means the statement belongs to a code instruction that executes a user-defined function. Hence, we must: 1) append to *seq* the rest of the statements of the aforementioned instruction, and 2) look for the bytecode for the user-defined function in *dict* to also append it to *seq*. Otherwise, the statement is simply appended to *seq*. All statements pass through the *ComposeStatement*, where the statement gets the desired information and format (i.e., the opcode or a list with the opcode and its parameters).

```
input  : A dictionary dict with the bytecode;
         The information info to include in the new dataset
output: The final sequence seq
seq ← [];
for $i \leftarrow 0, \ldots, len(\text{dict}['main']) - 1$ do
    if statement i is a call to a user-defined function then
        fname ← function name;
        while statement i belongs to the same instruction do
            seq.Append(ComposeStatement(statement i, info));
            $i \leftarrow i + 1$;
        end
        foreach statement ∈ dict [fname] do
            seq.Append(ComposeStatement(statement, info));
        end
    else
        seq.Append(ComposeStatement(statement i, info));
    end
end
return seq;
)
```

Algorithm 3: Get the desired information from a bytecode statement (*Get-Seq*

4.2 Network Construction and Model Evaluation

To construct the various network configurations and evaluate the resulting models, we implemented the Python script *model.py*. Algorithm 4 shows the overview of this script. The algorithm receives as input:

- the directory where the dataset that will be used to train and test the model is stored (*srcdir*);
- the directory where the collected metric values for each configuration will be stored (*dstdir*);
- a dictionary *hparam_dict* containing the values we want to experiment for each hyperparameter (hidden size, dropout rate, and learning rate);
- the file *base_voc* with the base vocabulary for the corresponding dataset.

It starts by executing the function *GetData*, which outputs the training and test sets $(X\_train, y\_train, X\_test, y\_test)$ and the base vocabulary. For each combination of hyperparameter values *hparam_grid*, the algorithm performs the 10-fold cross validation three times. Since the experiments with the OD did not use grid search, the configuration values were manually set instead of storing them in a dictionary. Next, the algorithm creates a dictionary *dict_test* where the evaluation metrics of each fold will be stored, and executes *ExecCV* (line 4), where the cross validation is performed. The function *ExecCV* is also responsible for outputting the updated *dict_test* with the performance of every fold. On line 5, we write the dictionary to a CSV file and store it in *dstdir*.

```
input  : A dictionary hparam_dict with all hyperparameter values to try A
         source directory srcdir with the dataset;
         A destination directory dstdir where the metrics results will be
stored;
         A file base_voc with the base vocabulary for the dataset
output: CSV files, each containing the metrics of each fold for a network
         configuration
1 $X\_train, y\_train, X\_test, y\_test, vocabulary \leftarrow$ GetData(srcdir,base_voc);
2 foreach $hparam\_grid \in$ ParameterGrid(hparam_dict) do
3     dict_test $\leftarrow \{accuracies' : [], 'precisions' : [], 'recalls' : []\}$;
4     dict_test $\leftarrow$ ExecCV(X_train, y_train, X_test, y_test, vocabulary,
        hparam_grid, dict_test);
5     FromDictToCVS(dict_test, GetCSVTestName(hparam_grid));
6 end
```

Algorithm 4: Construct, train, and test the DL network.

To understand better how we get the dataset, we provide the implementation of the function *GetData* (Algorithm 5), in which we prepare the train and test datasets. The algorithm receives the source directory *srcdir* where the dataset is stored, and the base vocabulary file *base_voc*. For each file in *srcdir*, we append its bytecode sequence to the list *docs* and its label (0 or 1) to the list *y* (lines 4 and 5, respectively). Next, we get the content of *base_voc* into the variable *vocabulary*. On line 10, we proceed to the split of the dataset (*docs*) into train and test. This function produces the stratified train test split mentioned in Section 5.1, in which the train and test sets keep the proportions between true and false labels equal to the original dataset, as indicated by the instruction on line 9.

```
   input  : A source directory srcdir with the dataset;
            A file base_voc with the base vocabulary for the dataset
   output: The dataset's base vocabulary vocabulary;
            The train and test datasets X_train, y_train, X_test, y_test
 1 docs ← [];
 2 y ← [];
 3 foreach file ∈ srcdir do
 4     docs.Append(sequence in file);
 5     y.Append(corresponding label for the sequence in file);
 6 end
 7 vocabulary ← base_voc content;
 8 test_size ← 0.3;
 9 stratify ← y;
10 X_train, y_train, X_test, y_test ← TrainTestSplit(docs, test_size, stratify);
11 return X_train, y_train, X_test, y_test, vocabulary;
```

Algorithm 5: Get the train and test data, and the base vocabulary (*Get-Data*)

Algorithm 6 shows the implementation of the function *ExecCV*. The algorithm starts by creating a data structure that stores the indices from *X_train* that will be used for train and test in each fold (line 3). Then, for each fold, we first obtain the actual train and test sets (*X_subtrain*,*y_subtrain*, *X_subtest*,*y_subtest*) by executing the function *TrainTestRSFKIndices* (line 5). In lines 6 to 8, we prepare the data to have the appropriate format for our model (i.e., numeric sequence where all entries have the same length). Thus, we completed the vocabulary with the remaining tokens from the train set *X_subtrain* (line 6). Afterwards, we transform the sequence into a numeric sequence following the preprocessing described in Chapter 5 (line 7), and we pad the data by checking the longest sequence size and adding '0' to the end of the others.

We create the model on line 9 with the desired configuration (maximum length of the input vectors, number of nodes for the LSTM layer, dropout rate and learning rate) and, on line 11, the model is trained for the (*X_subtrain*,*y_subtrain*) dataset. The batch size indicates how many training points are updated in an epoch. This value relates to the machine used and the granularity of the optimization algorithm. Although it can be tuned as a hyperparameter, we decided the value based only on the machine used, a 64-bit machine. To finish, on line 12 we execute *EvaluateFold*, which uses the trained model to predict the labels on the test set *X_subtest*. It then compares the predictions to the true labels *y_subtest*, and adds the corresponding evaluation metrics to *dict_test*.

```
input  : The train dataset X_train, y_train;
         A dictionary with the hyperparameter values hparam_dict;
         The base vocabulary vocabulary;
         The dictionary dict_test with the evaluation metrics for each fold
output: A set of CSV files with the test results for each fold and each test
         configuration;
         An update on dict_test, the dictionary with the evaluation metrics
         for each fold

1  $n\_splits \leftarrow 10$;
2  $n\_repeats \leftarrow 3$;
3  $rskf\_indices \leftarrow$ RepeatedStratifiedKFoldIndices ($n\_splits$, $n\_repeats$);
4  for $i \in n\_splits \times n\_repeats$ do
5      $X\_subtrain, X\_subtest, y\_subtrain, y\_subtest \leftarrow$
         TrainTestRSKFIndices(X_train, y_train, $i$);
6      vocabulary $\leftarrow$ GetCompleteVocabulary($X\_subtrain$, vocabulary);
7      $X\_subtrain, X\_subtest \leftarrow$ TokenToNumeric($X\_subtrain$, $X\_subtest$);
8      $X\_subtrain, X\_subtest, MAX\_LENGTH \leftarrow$ PadData($X\_subtrain$,
         $X\_subtest$);
9      model $\leftarrow$ CreateModel($MAX\_LENGTH$, hparam_dict ['hidden_size'],
         hparam_dict ['delta'], hparam_dict ['learning_rate']);
10     $batch\_size \leftarrow 64$;
11     model.Fit($X\_subtrain$, $y\_subtrain$, hparam_dict ['num_epochs'],
         $batch\_size$);
12     dict_test $\leftarrow$ EvaluateFold($X\_subtest$, $y\_subtest$, model, dict_test);
13 end
14 return dict_test;
```

Algorithm 6: Perform the cross validation for a configuration of hyperparameters (*ExecCV*)

The implementation of the function *CreateModel*, invoked by *ExecCV* (Algorithm 6), is provided in Algorithm 7. It creates a sequential model (line 1), to which it sequentially adds the model layers (lines 3, 5, 6, 8, and 9) with the given hyperparameters. The first layer added, the Embedding layer, keeps a matrix for the embedding vectors. So, the operands $VOCAB\_SIZE$ and *hidden_size* of the function *Embedding* indicate the matrix size; MAX_LENGTH indicates the layer's input length, whereas the boolean *trainable* taking the value *TRUE* indicates we want the embedding vectors to be learned along the other model's parameters (instead of being fixed). Then, we add the LSTM layer with input (and output) size equal to *hidden_size* and ask not to return the resulting sequences. For the experiments with 2 LSTM layers, the first LSTM layer would have this parameter set to *True*. Next, we add the Dropout layer and the two Dense layers with the appropriate sizes discussed in Section 3.2.2. Finally, on line 12, we compile the model using the loss function *binary_crossentropy* and the optimization algorithm RMSProp (initialized with the given learning rate). Again, for the first experiments with the OD, we also used other optimization algorithms. Here, we also define which metrics the model should track, which are *binary_accuracy*, *binary_precision*, and *binary_recall*. Section 5.1 provides further details on these metrics.

```
input  : The maximum length $MAX\_LENGTH$ for the input sequences;
         The vocabulary size $VOCAB\_SIZE$;
         The hidden size hidden_size;
         The dropout rate delta;
         The learning rate learning_rate
output: The parameterized model model
 1 model ← Sequential();
 2 trainable ← True;
 3 model.add(Embedding(VOCAB_SIZE, hidden_size, MAX_LENGTH,
     trainable));
 4 $return\_sequences \leftarrow False$;
 5 model.add(LSTM(hidden_size, return_sequences));
 6 model.add(Dropout(delta));
 7 $activation \leftarrow' relu'$;
 8 model.add(Dense(hidden_size, activation));
 9 model.add(Dense(1), Activation('sigmoid'));
10 $loss \leftarrow' binary\_crossentropy'$;
11 $metrics \leftarrow [binary\_accuracy, binary\_precision, binary\_recall]$;
12 model.compile(loss, RMSprop(learning_rate), metrics);
13 return model;
```

Algorithm 7: Implementation of the function *CreateModel*

Chapter 5

Experiments

In this chapter, we present the iterative experiments conducted to determine the final configuration of the DL network. We used two fundamentally different approaches for the experiment.

The first approach was a *manual* hyperparameter tuning, conducted with the Opcode Dataset (OD), which yielded starting values for the hyperparameters. We used those starting values for further experiments. Additionally, we also tested the architecture with three different optimization algorithms (ADADELTA, RMSProp, and Adam) and for 1 and 2 LSTM layers. The second approach was a grid search, which we used for the remaining datasets. We performed grid search employing an architecture composed of 1 LSTM layer and the RMSProp algorithm only, as it was under these configurations that the model performed better.

All experiments followed the proceedure mentioned in Section 5.1, with the 10-fold cross validation repeated three times on the training set. The accuracy, precision and recall metrics were also computed during the experiments.

5.1 How to evaluate the model

To evaluate the model, we applied a 70/30 random *stratified train-test split* to the dataset. Stratified operations maintain the proportion of vulnerable/non-vulnerable samples in each set, so both train and test sets resulting from the split will have the same proportion as the original dataset. Furthermore, we applied to the training set a *stratified 10-fold cross-validation* three times to each model. Applying this technique allows us to 1) train and validate each model 30 times on 70% of the data, and 2) test the final model on 30% of never-seen test data.

In classification tasks like the one we aim to solve, it is common to measure how good the model is at generalizing, by measuring its performance at a set of metrics. Let TP, TN, FP and FN be, respectively, the number of true positives, true negatives, false positives, and false negatives. We considered three well known metrics: accuracy (Equation 5.1), precision (Equation 5.2), and recall (Equation 5.3).

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

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

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

The accuracy indicates the rate of correctly predicted examples. Precision yields the proportion between correctly classified positive samples and positively classified samples. Recall provides the true positive rate, that is, the rate of positive examples correctly classified by the model. The three metrics are real numbers between 0 and 1. The higher the values of the three metrics, the more confident we are that the generalization power of the model is powerful. Normally, a better precision can be obtained at the cost of reducing the recall, so our goal is to balance the three metrics.

Since we train and validate each model 30 times, there will be 30 values for each metric, allowing us to make better-informed decisions: by having 30 values per metric, we can produce statistics that are more robust and trustworthy than a single value, which could easily be obtained by chance and lead us to faulty conclusions. This technique is even more relevant when working with a small dataset, such as ours, where the variance of the estimator is usually higher.

5.2 Evaluation with the Opcode Dataset

The experiments with the OD provided the first results of applying DL models to our vulnerability discovery task. To gain some practical understanding of the architecture and its appropriateness to the task, we decided to approach the experiments with the OD differently than we did with the remaining datasets. In practice, this translates to not optimizing the optimizer's hyperparameters default values in these first experiments (i.e., keeping them fixed). Table 5.1 lists these values.

Table 5.1: List of the default hyperparameters of each optimizer.

ADADELTA	RMSProp	ADAM
`lr` = 1	`lr` = 0.001	`lr` = 0.001
`rho` = 0.95	`rho` = 0.9	`beta_1` = 0.9
		`beta_2` = 0.999

In the following sections, we present the specifications we used to run the experiments for both the model with 1 LSTM and 2 LSTM layers.

5.2.1 Model with 1 LSTM layer

Configuration of the Experiments

Although we fixed the optimizer's hyperparameters, we decided to test other variables, namely:

- Number of units in the LSTM layer, $\mathit{HIDDEN\_SIZE}_1$ (since there is only one LSTM layer, let us denominate it by *HIDDEN_SIZE*). The number of units in the LSTM layer is strongly related to the model's learning capacity: the more units the layer has, the more we expect it to learn from the data. However, there is the usual trade-off: the more parameters a model has, the more likely it is to overfit the data, which results in the loss of generalization capacity by the model;

- Dropout rate, δ, a value between 0 and 1 associated with the Dropout Layer. It corresponds to the probability that a specific unit in the layer receives a 'corrupted' input: instead of receiving the expected input, the input vector is converted to a vector full of zeros. This process is called dropout, and its goal is to force the model to learn a good generalization of the data using as few parameters as possible;

- Number of epochs, which is the number of times the model cycles (i.e., updates its parameters) on the training set. Increasing the number of epochs usually results in a lower training loss. Once again, we need to be careful to tune this parameter, so that the model does not overfit on the training data.

We performed individual and separate manual tuning for these parameters. We also ran experiments for the three most suitable optimization algorithms for neural networks: ADADELTA, RMSProp, and ADAM. Table 5.2 shows the configurations tested for each possible parameter combination. For each parameter, we started by testing a wider range of values: $\{10, 20, 40, 80, 160\}$ for the *HIDDEN_SIZE* and number of epochs and $\{0.2, 0.3, 0.5\}$ for δ. Next, we fine-tuned the search by testing two values around the best result (one smaller and one greater), repeating the process as it seemed fit. Each configuration was tested for 10 epochs, except, of course, when tuning the number of epochs itself.

Results

According to the previously described approach, the results of each optimizer (trained with the best values found for each hyperparameter) are shown in Table 5.3. We can observe that ADADELTA is the optimizer with the worst performance. The values for the metrics are high and ADADELTA achieves higher values in two out of three

Table 5.2: Configurations tested for each hyperparameter.

Optimizer	HS	δ	NE
ADADELTA	10, 20, 40, 70 80, 90, 160	0.20, 0.30, 0.50	10, 20, 40, 70, 80, 90, 160, 200
RMSProp	10, 20, 40, 60 70, 75, 80, 85 90, 100, 160	0.15, 0.20, 0.25, 0.30, 0.50	10, 20, 40, 70, 80, 90, 160
ADAM	10, 20, 40, 65, 70, 75, 80, 90, 160	0.20, 0.25, 0.30, 0.35, 0.50	10, 20, 30, 35, 40, 45, 50, 70, 80, 90, 160

HS - HIDDEN_SIZE, δ - dropout rate, NE - number of epochs

Table 5.3: Results of the accuracy, precision and recall for the various configurations analysed.

Optimizer	HS	δ	NE	Accuracy	Precision	Recall
ADADELTA	80	0.30	160	0.9487	0.9837	0.9344
RMSProp	80	0.15	70	**0.9535**	0.9651	**0.9614**
ADAM	**70**	0.30	**35**	0.9413	**0.9876**	0.9189

HS - HIDDEN_SIZE, δ - dropout rate, NE - number of epochs

metrics - accuracy and recall - than ADAM, but it takes the longest to converge - it takes 160 epochs, against 70 epochs for RMSProp and 35 for ADAM. ADAM and RMSProp have the best results in different metrics: RMSProp has the best accuracy and recall, whereas ADAM has the best precision. Also note that ADAM can nearly achieve an accuracy as high as RMSProp (0.9413 against 0.9535) with only half the epochs (35 against 70) and less units (70 against 80). Since we are more interested in metrics' results (as opposed to computation cost), we consider that RMSProp achieves the best results under this hypothesis.

Figure 5.1 shows 9 box-plots that result from the tuning of RMSProp. The plots are organized in a 3 × 3 matrix, where each line corresponds to a hyperparameter, and each column to a metric. As we can see, the values of the box-plots tend to increase as the values of the horizontal axis increase, until they reach a point after which they start to decrease. There are a few machine learning concepts we should consider when analysing multiple box-pots and multiple metrics: i) although accuracy is the most intuitive metric, one's goal should always be to balance the performance of the three metrics; ii) it is important to consider not only the mean and median in each plot, but also the variance, represented by the height of the box; and iii) finally, when comparing two relatively similar box-plots, one should prefer the one that yielded fewer outliers. Based on these guidelines, we conclude that the best parameter selection for a 1 LSTM layer neural network is the following: **80 hidden units, 0.15 dropout rate and 70 epochs for training.**.

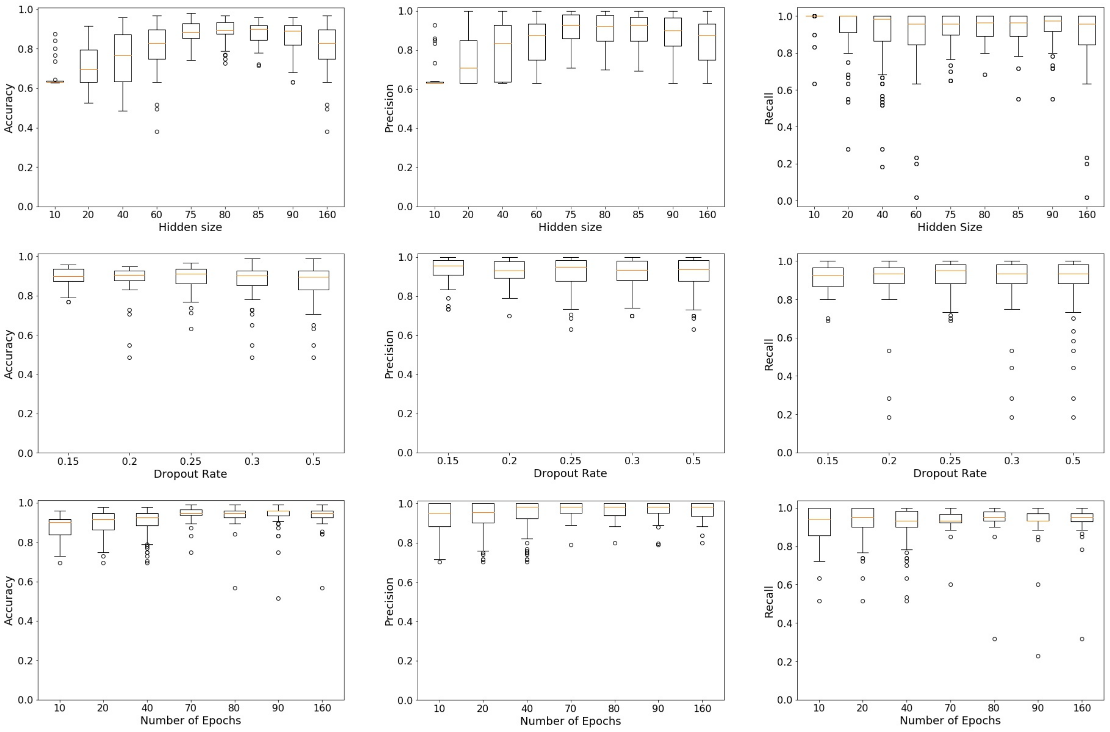


Figure 5.1: Box-plots for the hyperparameter tuning of the hidden size of the LSTM layer, dropout rate and number of epochs (respectively, the first, second and third rows), using the RMSProp optimizer.

5.2.2 Model with 2 LSTM layers

Configuration of the Experiments

Although the results for the 1-LSTM-layer model were good, we wanted to verify if it was possible to increase the model's performance by adding an extra LSTM layer. Therefore, we extended the experiments for a model with 2 LSTM+Dropout blocks. The experiment setting was similar to the previous one, but we added two new hyperparameters for the new layers. The new model's hyperparameters are: $HIDDEN\_SIZE_1$, $HIDDEN\_SIZE_2$, δ_1, δ_2, and the number of epochs. Once more, we used manual hyperparameter tuning by firstly tuning $HIDDEN\_SIZE_1$ and $HIDDEN\_SIZE_2$, then δ_1 and δ_2, and finally the number of epochs. From now on, we will only resort to the RMSProp optimizer because it produced the best results for the 1-LSTM-layer model.

Table 5.4 shows the values tested for each hyperparameter. Considering that the deeper a layer is the more specialized it is, the chosen values for $HIDDEN\_SIZE_2$ are higher than the values chosen for $HIDDEN\_SIZE_1$. That is, we expect the second LSTM layer to learn finer details than the first one, thus it needs more neurons. On the other hand, for the dropout rates δ_1 and δ_2 it is the opposite: the deeper we go into the model the less we want to ”forget” because we are working with representations already learned by the model.

Table 5.4: Configurations tested for each hyperparameter for the OD using RSMP.

Hyperparameter	Values
$HIDDEN\_SIZE_1$	35, 40, 45
$HIDDEN\_SIZE_2$	75, 80, 85
Number of Epochs	10, 20, 40, 70, 80, 90
δ_1	0.35, 0.40, 0.45
δ_2	0.05, 0.1, 0.15

Results

Analogously to the box-plots presented in the previous subsection, Figure 5.2 shows a 3×3 matrix of box-plots corresponding to the experiments executed for the 2-LSTM-layer model. Each row depicts the model's performance for each hyperparameter set, being the first row for $(HIDDEN\_SIZE_1, HIDDEN\_SIZE_2)$, the second for (δ_1,δ_2), and the third for the number of epochs. The columns correspond to the box-plots for the accuracy, precision, and recall, respectively.

By using the same criteria followed in the previous subsection, we choose the following values as the ones leading to the best performance:

- (45, 75) for the pair $(HIDDEN\_SIZE_1, HIDDEN\_SIZE_2)$;
- (0.45, 0.05) for the pair (δ_1, δ_2);
- 80 for the number of epochs.

To compare the 1-LSTM-layer model with the 2-LSTM-layer model, we trained the whole training set with the best configuration of each model and tested the result on the test set. Table 5.5 shows the results obtained, for both models, for the training and the testing. The first thing we notice is that, since the test values are just slightly worse than the train values, there is no overfitting. We also note that both models' performance is also quite similar, especially regarding the accuracy. The 1-LSTM-layer model has a considerably better precision (more or less 4% better), while the recall is worse (more or less 3% worse). Therefore, we can conclude that the addition of an LSTM+Dropout block did not significantly improve the model's performance. Besides, the training time is consideratelly superior, which leads us to prefer the 1-LSTM-layer model. In the following experiments, we focus our study in a model with a single 1 LSTM+Dropout block.

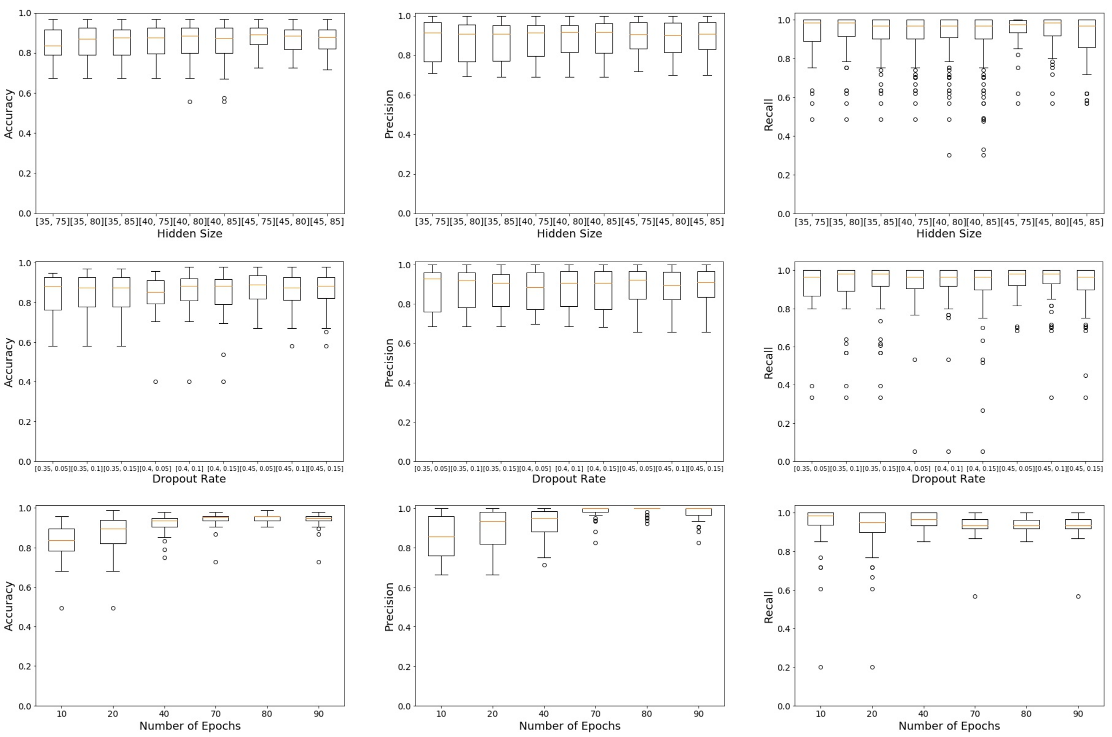


Figure 5.2: Box-plots for the hyperparameter tuning of the hidden sizes of the 2 LSTM layers, dropout rates for the 2 Dropout layers and number of epochs (respectively, the first, second and third rows), using the RMSProp optimizer.

Table 5.5: Performance of the best configurations in both the train and test sets for the 1-LSTM and 2-LSTM models.

	Train			**Test**		
Model	Accuracy	Precision	Recall	Accuracy	Precision	Recall
1-LSTM-layer	0.9559	**0.9701**	0.9605	**0.9535**	**0.9614**	0.9651
2-LSTM-layer	**0.9570**	0.9336	**0.9982**	0.9438	0.9189	**0.9917**

5.3 Evaluation with the remaining datasets

5.3.1 Configuration of the Experiments

After the experiments with the OD, we opted to automatize the remaining experiments with the Opcode+Operand Dataset (OOD), the Slice Dataset (SD), and Simplified Slice Dataset (SSD), by applying grid search. With grid search, we define the values of the hyperparameters we want to test, and evaluate all possible combinations. We only tested the 1-LSTM layer model because increasing the number of hidden layers did not substantially improve the model's performance, and the training and tuning were too time-consuming. Instead, we decided to add the learning rate to the list of tunable hyperparameters. The learning rate is a crucial hyperparameter regarding learning speed and efficiency. In a first approach, it seemed

Table 5.6: Configurations tested for each hyperparameter with OOD, SD, and SSD, using RMSProp.

Hyperparameters	Values
HIDDEN_SIZE	10, 15, 20, 30
Learning Rate	0.01, 0.001
Number of Epochs	60, 80, 100
δ	0.1, 0.2, 0.3

sufficient to use the predefined value, but to obtain better results, we decided to introduce it for the rest of the experiments.

Table 5.6 shows the hyperparameters that were tuned and their values. As it shows, these are similar as in the first experiments with the addition of the learning rate.

5.3.2 Results

Tables 5.7, 5.8, and 5.9 exhibit the performance of each tested configuration for the OOD, SD, and SSD, respectively. These contain the mean value for each metric, obtained by cross-validating each configuration. The best values for each dataset are displayed in bold.

From the results, it is apparent that the addition of the opcode's operands did not improve the model's performance. The best configuration achieved around 64% of accuracy and precision and almost 100% of recall. Although recall is almost perfect (meaning the model predicts very well positive examples), the values for the other metrics are considerably worse compared to the results for the OD. These results are not unexpected since the dimension of the input space got higher, increasing the problem's complexity. However, we cannot dismiss the importance of including operands in our model. Operands, as we saw in Chapter 3, play a crucial role in detecting SQLi.

The results for the SD and SSD datasets confirm that, by linearizing the excerpts, the model can learn to classify them easier. Our approach led to an improvement of over 10% in accuracy and precision. We consider that the fact that the recall got worse values does not impose an issue as the balance between the metrics is higher than for the model trained with the OOD. It is still important to assert how the model would behave if it included all original examples. As we saw in Section 3.3, SD was manually created and, even though the final dataset contains more samples than BED, these correspond to a small part of those samples.

To finish, the results to the SSD show a slight deterioration compared to the SD results, which leads us to question the validity of the simplification approach. Nevertheless, it is not wise to dismiss the opcode grouping approach before testing

Table 5.7: Performance of the model for the OOD in every configuration tested (HS - hidden size, δ - dropout rate, LR - learning rate, NE - number of epochs). The metrics presented are the mean value for the 30 folds run (Acc - accuracy, Prec - precision, Rec - recall).

HS											
10—						15					
δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**	δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**
0.1	0.01	60	0.6331	0.6326	1.0000	0.1	0.01	60	0.6174	0.5918	0.9328
		80	0.6334	0.6329	0.9994			80	0.6345	0.6335	1.0000
		100	0.6334	0.6330	0.9989			100	0.6262	0.6127	0.9667
	0.001	60	0.6327	0.6324	1.0000		0.001	60	0.6261	0.6128	0.9661
		80	0.6341	0.6333	1.0000			80	0.6345	0.6335	1.0000
		100	0.6162	0.5910	0.9328			100	0.6262	0.6128	0.9661
0.2	0.01	60	0.6327	0.6324	1.0000	0.2	0.01	60	0.6331	0.6335	0.9944
		80	0.6341	0.6333	0.9994			80	0.6278	0.6146	0.9628
		100	0.6338	0.6332	0.9989			100	0.6191	0.5928	0.9328
	0.001	60	0.6331	0.6326	1.0000		0.001	60	0.6327	0.6324	1.0000
		80	0.6334	0.6329	0.9994			80	0.6362	0.6346	1.0000
		100	0.6341	0.6334	0.9994			100	0.6296	0.6155	0.9633
0.3	0.01	60	0.6320	0.6320	0.9994	0.3	0.01	60	0.6327	0.6324	1.0000
		80	0.6331	0.6327	0.9995			80	0.6338	0.6333	0.9989
		100	0.6334	0.6328	1.0000			100	0.6355	0.6342	1.0000
	0.001	60	0.6327	0.6324	1.0000		0.001	60	0.6331	0.6331	0.9972
		80	0.6243	0.6116	0.9661			80	0.6247	0.6118	0.9667
		100	0.6324	0.6322	0.9995			100	0.6341	0.6333	1.0000
HS											
20						30					
δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**	δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**
0.1	0.01	60	0.6243	0.6115	0.9667	0.1	0.01	60	0.6261	0.6127	0.9667
		80	0.6254	0.6130	0.9661			80	0.6303	0.6156	0.9656
		100	0.6295	0.6150	0.9661			100	0.6436	0.6400	0.9972
	0.001	60	0.6258	0.6126	0.9661		0.001	60	0.6261	0.6128	0.9661
		80	0.6262	0.6128	0.9661			80	0.6369	0.6352	0.9994
		100	0.6262	0.6130	0.9656			100	0.6156	0.5757	0.8972
0.2	0.01	60	0.6331	0.6326	1.0000	0.2	0.01	60	0.6247	0.6118	0.9667
		80	0.6285	0.6143	0.9667			80	0.6306	0.6161	0.9645
		100	0.6362	0.6353	0.9961			100	0.6225	0.5958	0.9289
	0.001	60	0.6352	0.6340	0.9994		0.001	60	0.6345	0.6335	1.0000
		80	0.6345	0.6335	1.0000			80	0.6201	0.5934	0.9333
		100	0.6362	0.6348	0.9989			100	0.6439	0.6400	0.9977
0.3	0.01	60	0.6331	0.6326	1.0000	0.3	0.01	60	0.6338	0.6330	1.0000
		80	0.6341	0.6333	1.0000			80	0.6366	0.6354	0.9967
		100	0.6348	0.6341	0.9978			100	**0.6443**	**0.6407**	**0.9956**
	0.001	60	0.6076	0.5705	0.8978		0.001	60	0.6345	0.6336	0.9994
		80	0.6093	0.5710	0.9000			80	0.6352	0.6346	0.9961
		100	0.6366	0.6348	1.0000			100	0.6429	0.6393	0.9984

Table 5.8: Performance of the model for the SD in every configuration tested (HS - hidden size, δ - dropout rate, LR - learning rate, NE - number of epochs). The metrics presented are the mean value for the 30 folds run (Acc - accuracy, Prec - precision, Rec - recall).

HS											
10						15					
δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**	δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**
0.1	0.01	60	0.6510	0.6079	0.6790	0.1	0.01	60	0.6782	0.6690	0.7389
		80	0.7243	0.6901	0.7710			80	0.7171	0.6905	0.7593
		100	0.7252	0.6866	0.8210			100	0.7411	0.7065	0.7975
	0.001	60	0.6370	0.6134	0.6142		0.001	60	0.7018	0.6571	0.7883
		80	0.6993	0.6545	0.7969			80	0.7365	0.6743	0.7944
		100	0.6972	0.6675	0.7920			100	0.7296	0.6841	0.8457
0.2	0.01	60	0.6495	0.6199	0.6420	0.2	0.01	60	0.6750	0.6444	0.7253
		80	0.6964	0.6549	0.7877			80	0.7034	0.6592	0.7377
		100	0.7098	0.6719	0.7883			100	0.7397	0.7074	0.8148
	0.001	60	0.6424	0.6437	0.6765		0.001	60	0.6713	0.6428	0.7525
		80	0.6691	0.6310	0.7852			80	0.7198	0.6414	0.7938
		100	0.7157	0.6614	0.8636			100	0.7078	0.6686	0.7753
0.3	0.01	60	0.6461	0.6200	0.7154	0.3	0.01	60	0.6790	0.6389	0.7457
		80	0.6889	0.6446	0.7846			80	0.7267	0.6900	0.7704
		100	0.6900	0.6467	0.7852			100	0.7431	0.7001	0.8272
	0.001	60	0.6531	0.6244	0.6840		0.001	60	0.6860	0.6378	0.7302
		80	0.6981	0.6529	0.7679			80	0.7224	0.6843	0.8000
		100	0.7245	0.6945	0.7710			100	0.7356	0.7030	0.8173

HS											
20						30					
δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**	δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**
0.1	0.01	60	0.7220	0.6667	0.8642	0.1	0.01	60	0.6857	0.6320	0.6920
		80	0.7211	0.7148	0.7796			80	0.7324	0.7068	0.7290
		100	0.7411	0.7061	0.8198			100	0.7838	0.7824	0.7988
	0.001	60	0.7107	0.6471	0.7944		0.001	60	0.7230	0.6581	0.7525
		80	0.7247	0.7047	0.7457			80	0.7224	0.7237	0.6846
		100	0.7573	0.7389	0.8111			100	0.7633	0.7845	0.7210
0.2	0.01	60	0.6883	0.6391	0.7346	0.2	0.01	60	0.6989	0.7105	0.6846
		80	0.7178	0.6993	0.7074			80	0.7452	0.7294	0.7735
		100	0.7613	0.6943	0.8500			100	0.7667	0.7924	0.7426
	0.001	60	0.7154	0.6549	0.7191		0.001	60	0.6840	0.6730	0.7210
		80	0.7426	0.7406	0.7543			80	0.7408	0.7038	0.7704
		100	0.7493	0.7237	0.8012			100	0.7789	0.7857	0.7667
0.3	0.01	60	0.7081	0.6673	0.7475	0.3	0.01	60	0.7020	0.6845	0.7444
		80	0.7371	0.7073	0.8130			80	0.7289	0.7053	0.7117
		100	0.7024	0.6377	0.7716			100	0.7616	0.7489	0.7907
	0.001	60	0.7051	0.6547	0.8228		0.001	60	0.7054	0.6546	0.7105
		80	0.7326	0.7013	0.8006			80	0.7529	0.7350	0.7852
		100	0.7339	0.7117	0.7809			100	**0.7783**	**0.7721**	**0.7969**

other grouping configurations. It may be that we are losing vital information we did not consider to be relevant at first.

Table 5.10 shows the final performance of the model when trained with each dataset with the whole training data. The last three columns present the values for the test set after training. Similarly to the previous results, the model performed better with the SD, in which the metrics are more balanced. Accuracy and precision are the highest of the three. Recall is the worst, but is still very good (above 80%) and we consider the overall performance as being the best.

Summing up, we experimented five different approaches: using the OD with 1 LSTM+Dropout block, the OD with 2 LSTM+Dropout blocks, training with the OOD, SD, and SSD. The best results occurred for the first case. Since it is not very realistic that opcodes alone can be used to classify excerpts regarding SQLi vulnerability, we believe the best approach is using the SD. A central issue that affects the quality of these results is the simplicity of the original dataset. The generated test cases extracted from SARD do not reflect well the web application paradigm as samples are small and very similar to each other. Nevertheless, there are no other open and ready to use datasets. It is our understanding that it is crucial to create a more suitable original dataset to verify our conclusions.

Table 5.9: Performance of the model for the SSD in every configuration tested (HS - hidden size, δ - dropout rate, LR - learning rate, NE - number of epochs). The metrics presented are the mean value for the 30 folds run (Acc - accuracy, Prec - precision, Rec - recall).

HS											
10						15					
δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**	δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**
0.1	0.01	60	0.6268	0.6050	0.6704	0.1	0.01	60	0.6458	0.6340	0.6562
		80	0.6610	0.5966	0.6401			80	0.7097	0.7019	0.7586
		100	0.7091	0.6787	0.7611			100	0.7397	0.6838	0.7771
	0.001	60	0.6339	0.5857	0.6105		0.001	60	0.6802	0.6448	0.7525
		80	0.6901	0.6554	0.7364			80	0.6967	0.6697	0.7253
		100	0.6974	0.6589	0.7759			100	0.7074	0.6949	0.6802
0.2	0.01	60	0.6052	0.5797	0.6272	0.2	0.01	60	0.6506	0.6154	0.6605
		80	0.6548	0.6217	0.7229			80	0.6973	0.6759	0.7340
		100	0.6941	0.6350	0.8549			100	0.7273	0.6985	0.8049
	0.001	60	0.6393	0.5957	0.6346		0.001	60	0.6464	0.6183	0.7623
		80	0.6465	0.6358	0.6630			80	0.7152	0.6729	0.7494
		100	0.6762	0.6132	0.7123			100	0.7472	0.7241	0.7914
0.3	0.01	60	0.5925	0.5179	0.5253	0.3	0.01	60	0.6750	0.6341	0.8315
		80	0.6364	0.6068	0.6765			80	0.6929	0.6890	0.7259
		100	0.6737	0.6590	0.7475			100	0.7184	0.7141	0.7500
	0.001	60	0.6545	0.6051	0.6710		0.001	60	0.6724	0.6290	0.7265
		80	0.6479	0.6214	0.6710			80	0.7060	0.6643	0.7877
		100	0.6569	0.6472	0.6636			100	0.7290	0.6578	0.7759

HS											
20						30					
δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**	δ	**LR**	**NE**	**Acc**	**Prec**	**Rec**
0.1	0.01	60	0.7071	0.6615	0.7432	0.1	0.01	60	0.6875	0.6781	0.7049
		80	0.7107	0.6929	0.6920			80	0.7101	0.7433	0.6185
		100	0.7256	0.7313	0.7333			100	0.7386	0.7646	0.6722
	0.001	60	0.7117	0.6920	0.7185		0.001	60	0.6727	0.6300	0.6407
		80	0.7429	0.7239	0.7580			80	0.7422	0.7537	0.7302
		100	**0.7471**	**0.7231**	**0.8043**			100	0.7649	0.8059	0.7086
0.2	0.01	60	0.6756	0.6514	0.7321	0.2	0.01	60	0.7033	0.7062	0.6432
		80	0.7148	0.7031	0.7611			80	0.7151	0.7281	0.7111
		100	0.7614	0.7752	0.7370			100	0.7495	0.7536	0.7698
	0.001	60	0.6759	0.6327	0.6914		0.001	60	0.7013	0.7021	0.6784
		80	0.7331	0.6839	0.8259			80	0.6964	0.6841	0.6463
		100	0.7357	0.7378	0.7642			100	0.7477	0.7439	0.6852
0.3	0.01	60	0.6927	0.6830	0.7160	0.3	0.01	60	0.7028	0.6788	0.6864
		80	0.7373	0.7114	0.7846			80	0.7002	0.6998	0.6012
		100	0.7291	0.7270	0.6870			100	0.7580	0.7495	0.7340
	0.001	60	0.6788	0.6417	0.6850		0.001	60	0.6886	0.7301	0.6080
		80	0.7304	0.6995	0.7826			80	0.7238	0.7667	0.6556
		100	0.7364	0.7210	0.7096			100	0.7420	0.7481	0.72351

Table 5.10: Performance of the best configurations in both the train and test sets for the 1-LSTM and 2-LSTM models.

	Train			**Test**		
Dataset	Accuracy	Precision	Recall	Accuracy	Precision	Recall
OOD	0.6406	0.6370	**1.0000**	0.6399	0.6355	**0.9994**
SD	**0.7926**	0.**7295**	0.8840	0.**7811**	0.**7296**	0.8455
SSD	0.7762	0.7030	0.9024	0.7590	0.6902	0.8798

Chapter 6

Conclusion

The developed work is a starting point for research on the use of DL models for the discovery of web vulnerabilities, where code is represented in an IL. We focused on SQLi vulnerabilities that are the most common and damaging web vulnerabilities. We resorted to a DL architecture suitable for NLP by leveraging similarities between bytecode and natural language. The model is composed of an Embedding layer, followed by n blocks of LSTM+Dropout layers, and finishes with two Dense layers, that work sequentially.

Initially, we planned on focusing the research on model's architectures. However, throughout the process, we noticed it was a good contribution to evaluate which type of data to feed the model. Hence, we created several datasets that represent the original PHP bytecode excerpts differently:

- the Opcode Dataset (OD), which represents excerpts as sequences of opcodes;
- the Opcode+Operand Dataset (OOD), that represents a bytecode excerpt by their opcodes and operands;
- the Slice Dataset (SD), where we managed to separate each bytecode excerpt into slices (execution paths) to obtain linear control flow;
- the Simplified Slice Dataset (SSD), similar to SD but with less unique opcodes, and therefore, with fewer input features.

We ran a set of experiments on each dataset, by performing cross-validation on different configurations of the model's hyperparameters. The OD achieved the best results, with more than 90% on all metrics. SD and SSD also performed well, with results above 70%. The dataset that led to worse results was the OOD. We believe the results support the validity of our slice approach over the OOD. Even though OD got the best results, this is most likely due to the vocabulary's size versus number of available samples. It is essential to conduct other experiments where, for instance, we expand PED.

6.1 Future Work

This work can lead to a series of further research, such as:

- Building datasets from existing web applications. First, it is crucial to test our model against such datasets to assert its performance with more complex and realistic data. Then, it would be pertinent to retrain the model with some of these datasets to raise the quality of the model and its usability in practice.

- Studying more complex DL architectures that incorporate different types of layers suitable for NLP tasks (e.g., transformers, LSTM+CNN). Our research did not focus much on the model construction, so it is probable that other architectures with different layers known to work well in NLP could improve the model's performance.

- The use of the model to find other web vulnerabilities, such as Cross-site Scripting. Although SQLi is the most common web vulnerability, there are other web vulnerabilities that impose serious threats to web applications. It would be interesting to see how to extend the model to other web vulnerabilities and compare the viability of having one general model for all types of bugs against having different specialized models, one for each type of web vulnerability.

- Investigating the slice approach with more data. In our work, we converted excerpts into slices manually, which was too time consuming and did not allow us to convert all available excerpts. It is essential to automate the converting task to increase the dataset's size and compare both approaches properly.

- Investigating other grouping options for the opcodes and operands to assert the approach's viability.

Bibliography

[1] C. C. Aggarwal et al. *Neural Networks and Deep Learning*. Springer, 2018.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, techniques and tools. *Addison Wesley*, 7(8):9, 1986.

[3] F Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[4] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[5] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] T. Ben-Nun, A. Jakobovits, and T. Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, pages 3585–3597, 2018.

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

[8] M. Boden. A guide to recurrent neural networks and backpropagation. *The Dallas Project*, 2002.

[9] Time Google Brain. Tensorflow. `https://www.tensorflow.org/`, 2015.

[10] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, pages 1724–1734, 2014.

[11] F. Chollet. *Deep Learning with Python*. Manning Publications Company, 2017.

[12] F. Chollet et al. Keras. `https://keras.io`, 2015.

[13] J. Clarke-Salt. *SQL Injection Attacks and Defense*. Elsevier, 2009.

[14] M. Correia and P. Sousa. Segurança no software. *Lisboa: FCA*, 2010.

[15] J. Devlin, M. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[16] J. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[17] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[18] A. Fidalgo, I. Medeiros, P. Antunes, and N. Neves. Towards a deep learning model for vulnerability detection on web application variants. In *2020 Workshop on Testing of Configurable and Multi-variant Systems co-located with the 2020 IEEE International Conference on Software Testing*, pages 465–476. IEEE, 2020.

[19] Y. Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017.

[20] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* MIT press, 2016.

[21] G. Grieco, G. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*, pages 85–96. ACM, 2016.

[22] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and Xi. Xing. Lemna: Explaining deep learning dased security applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379. ACM, 2018.

[23] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[24] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[25] D. Kingma and J. Ba. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[26] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[27] J. Kronjee. Discovering vulnerabilities using data-flow analysis and machine learning. Master's thesis, Open Universiteit Nederland, 2018.

[28] O. Levy, Y. Goldberg, and I. Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[29] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 10(5):1692, 2020.

[30] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *arXiv preprint arXiv:2001.02350*, 2020.

[31] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756*, 2018.

[32] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[33] M. Luong, H. Pham, and C. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[34] I. Medeiros, N. Neves, and M. Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, 2015.

[35] I. Medeiros, N. Neves, and M. Correia. Statically detecting vulnerabilities by processing programming languages as natural languages. *arXiv preprint arXiv:1910.06826*, 2019.

[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[37] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association*, volume 2, pages 1045–1048, 2010.

[38] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of Advances in Neural Information Processing Systems*, pages 3111–3119, 2013.

[39] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[40] Karl Joseph Ottenstein. Data-flow graphs as an intermediate program form. 1978.

[41] PHP. Php supported versions. `https://secure.php.net/supported-versions.php`, 2019.

[42] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications*, pages 757–762, 2018.

[43] M. Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[44] T. J Sejnowski and Charles R. R. Parallel networks that learn to pronounce english text. *Complex Systems*, 1(1):145–168, 1987.

[45] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[46] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[47] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pages 409–415, 2016.

[48] I. Sutskever, O. Vinyals, and Q. Le. Sequence to sequence learning with neural networks. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[49] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2):26–31, 2012.

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[51] W3Techs. Usage statistics of php for websites. `https://w3techs.com/technologies/details/pl-php`, 2019.

[52] J. Williams and D. Wichers. Top 10-2017 the ten most critical web application security risks. *URL: owasp. org/images/7/72/OWASP_Top_10-2017_% 28en*, 29, 2017.

[53] Y. Wu, M. Schuster, Z. Chen, Q. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[54] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, pages 13–13, 2011.

[55] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[56] M. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[57] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207, 2019.