Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants

Ana Fidalgo, Ibéria Medeiros, Paulo Antunes, and Nuno Neves
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
afidalgo@lasige.di.fc.ul.pt, imedeiros@di.fc.ul.pt, pdantunes@fc.ul.pt, nuno@di.fc.ul.pt

***Abstract*—Reported vulnerabilities have grown significantly over the recent years, with SQL injection (SQLi) being one of the most prominent, especially in web applications. For these, such increase can be explained by the integration of multiple software parts (e.g., various plugins and modules), often developed by different organizations, composing thus *web application variants*. Machine Learning has the potential to be a great ally on finding vulnerabilities, aiding experts by reducing the search space or even by classifying programs on their own. However, previous work usually does not consider SQLi or utilizes techniques hard to scale. Moreover, there is a clear gap in vulnerability detection with machine learning for PHP, the most popular server-side language for web applications. This paper presents a Deep Learning model able to classify PHP slices as vulnerable (or not) to SQLi. As slices can belong to any variant, we propose the use of an intermediate language to represent the slices and interpret them as text, resorting to well-studied Natural Language Processing (NLP) techniques. Preliminary results of the use of the model show that it can discover SQLi, helping programmers and precluding attacks that would eventually cost a lot to repair.**

***Index Terms*—web application vulnerabilities, vulnerability detection, natural language processing, deep learning, software security.**

I. Introduction

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 appealing to hackers, and the number of vulnerabilities has continuously grown year by year. Reported vulnerabilities have significantly grown over the recent years [1], with SQL injection (SQLi) being one of the most prominent, especially in web applications, and appearing on Top 10 of OWASP [2]. Furthermore, SQLi is considered to be one of the most devastating web vulnerabilities, as they allow intruders to access private data, and successful attacks cost companies much money for repairs. Also, SQLi vulnerabilities are relatively easy to exploit, making them even more appealing to attackers. For example, they represent two-thirds (65%) of cyberattacks of all web attacks [3].

Such increase of SQLi vulnerabilities can also be explained by the integration of multiple software parts (e.g., various plugins and modules), often developed by different organizations, composing thus *web application variants*. For instance, WordPress plugins are well-known for their bad security shape, as shown on works of Nunes et al. [4] and Medeiros et al. [5].

Several tools employ different techniques for detecting web application vulnerabilities, being the most popular those that rely on static analysis [6]–[9], which searches vulnerabilities by analyzing the source code of the applications, and fuzzing [10]–[13], which performs attacks against a target application by injecting malicious inputs and verifies if they are succeeding well (i.e., exploiting any vulnerability existent in the application). However, both techniques have presented limitations which underling the production of false positives (inexistent vulnerabilities) and false negatives (vulnerabilities not detected), respectively. Recently, Machine Learning (ML) has emerged as a technique to discover vulnerabilities. Some tools resort to data mining [14], [15], which extract features from the source code of the applications that can be related to vulnerabilities and predict if applications contain vulnerabilities, while others employ Natural Language Processing (NLP) algorithms [16] and neural networks [17], [18] to process the application code, and output if it contains flaws. Although the advances made with ML models have made the first steps on accurate and precise detection of vulnerabilities, there is still space for improvements in this field.

In this paper, we propose a novel approach leveraging Deep Learning (DL) and NLP to classify PHP slices as vulnerable (or non-vulnerable) to SQLi. Although PHP is the most popular server-side language for web applications [19], the detection of software vulnerabilities in PHP, through ML, is still relatively unexplored. We process PHP code slices in their opcode format that is similar to the Assembly opcodes of C/C++, and is considered an intermediate language. This approach allows us to look closer to the internal structure of the language, which can help in the classification task we aim to solve. Moreover, as slices can belong to any *web application variant*, the use of an intermediate language to represent them can facilitate the analysis transversality between variants. Furthermore, this intermediate language has not yet been used to solve our task.

We use DL and NLP state-of-the-art methodologies to propose a model to solve the task. We start by using DL models, such as Long Short-Term Memory (LSTM) layers, that take into consideration the context of each opcode. These are commonly employed to solve NLP tasks and were used in some works to detect vulnerabilities in other programming languages [20] [17] [21]. However, to the best of our knowledge, they were never used to process neither PHP code or opcode.

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00083

In the model we propose, the vulnerability detection process starts by representing each PHP opcode as an embedding vector that is trained along the rest of the parameters. Next, the model uses an LSTM, Dropout, and Dense layers, outputting the probability of the slice being vulnerable to SQLi. Our approach uses a dataset 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 slice that starts in an entry point (instruction that receives user-defined input, like *$_GET*) and finishes in a sensitive sink (a function, like *mysqli_query*, that, when executed with malicious input, may cause undesired behavior, such as giving access to private data to an unauthorized person).

We evaluate the model with various hyperparameter configurations for different DL optimizers, and the results showed that the model can discover SQLi, helping programmers and precluding attacks that would eventually cost a lot to repair.

The contributions of the paper are: 1) the analysis of PHP web applications in the intermediate PHP opcode language, 2) a DL model that accurately classifies PHP opcode slices as SQLi vulnerable or non-vulnerable, 3) a dataset of PHP opcode slices, 4) pre-trained embedding vectors for each PHP opcode, and 5) an experimental evaluation providing different assessments of different hyperparameter configurations.

The outline of this paper is as follows. Section II introduces some background information on vulnerabilities and DL in the context of NLP. In Section III, a comparison with previous work and ours is made. In Section IV, the methods we use, the dataset we composed, the model and how to evaluate it are presented. Section V presents the experimental results for the models already investigated, and Section VI concludes the paper.

II. Background

A. Web Vulnerabilities

Vulnerabilities are flaws present in a system. When an attacker exploits them, he can breach some security policy, and their impact can cost a significant amount of money to the organization. Over the last few years, with the increasing importance of the Internet and web applications being widely used, the number of vulnerabilities has grown exponentially [22]. According to the OWASP Top 10 of 2017 [2], 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

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

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 code as an argument to be included in a query. Injections are easy to exploit - the attacker must only insert appropriate strings to exploit the target interpreter, usually through the addition of meta characters -, and have a high impact on the system, making them 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 [9].

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 [23], SQLi is one of the most devastating vulnerability classes. 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 vulnerability.

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 [19], 79% of web applications are written in PHP.

```
1 $tainted = $_SESSION['UserData'];
2
3 if (filter_var($tainted, FILTER_VALIDATE_EMAIL))
4   $t = $tainted ;
5 else
6   $t = "" ;
7
8 $query = sprintf("SELECT * FROM '%s'", $t);
9 $conn = mysql_connect('localhost', 'user', 'pass');
10 mysql_select_db('dbname');
11 $res = mysql_query($query);
```

Listing 1: Example of a PHP slice vulnerable to SQLi.

The PHP example depicted in Listing 1 is an adaptation of a SARD vulnerable sample. On line 1, exterior data enters the program through the global array `$_SESSION`, and it is stored in the `$tainted` variable. Next, the `$tainted` value passes through a filter that indicates whether its format is according to the *FILTER_VALIDATE_EMAIL* (line 3). This filter, however, does not escape characters like *"'"* and *" "*. An attacker could, for example, give as input a string in the format *"'<malicious query>'@gmail.com"*, where *<malicious query>* would be anything following the SQL syntax. On line 8, a query string is constructed with the given input. This way, the attacker might get access to private data stored in `$res` (line 11),

which stores the result from the execution of the query in the database. Therefore, the slice has a SQLi vulnerability. This vulnerability can be fixed by escaping special characters besides validating the email format.

B. 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. Nonetheless, 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, but 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 the previous layer and applies some transformation to the input. By having multiple layers, the model can learn more complex and useful concepts, based on simpler and broader ones (from previous layers) [24].

DL models, like any ML model, have an implicit cost function they intend to minimize, i.e., they are minimization tasks for which they need an optimization algorithm. Normally, neural network optimizers are based on the Stochastic Gradient Descent (SGD). There are three SGD-based optimizers that are frequently used: 1) ADADELTA [25], which uses a dynamic learning rate (`lr`) computed per-dimension, 2) RMSProp [26], which generates updates using a re-scaled gradient, and 3) ADAM [27] that generates updates with a running average of the gradient.

NLP deals with natural language data. Because of the data characteristics, namely lack of structure, ambiguity, discreteness, and sparseness [28], it is often hard to find a suitable representation for the desired task, making DL methods very popular in the field [29], [30].

There are two widely used components in DL for NLP: Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) [28]. 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 all past words in a sequence. They are excellent for sequential data and have achieved state-of-the-art results in many NLP fields [31], [32].

CNN and RNN layers are often preceded by an embedding layer, which maps discrete symbols into continuous vectors, solving the sparsity problem of natural language data. Also, it is common to feed the output of these components to a feed-forward component that learns to perform the desired task, like classification [28].

Recently, some new approaches have achieved exciting results, such as the combination of attention and RNN or CNN [33], [34] and attention only, with resort to transformers [35], [36].

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. Each time step, the window slides, until covering the whole sequence, and produces a scalar value that represents the tokens from that time window [28]. We can apply n filters to each window, which results in an n dimensional vector that characterizes that window. Finally, the resulting vectors are combined through a *pooling operation* into a single vector, that represents the whole sequence. There are several pooling operations, however, 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, keeps the k highest values, preserving their order, and concatenates the resulting vectors.

CNN cannot, however, extract the global order of the input - only local order can be represented. Hence, this architecture is specially good in solving computer vision tasks, like image classification and object recognition [37], [38].

2) Recurrent Neural Networks: RNNs are networks that maintain a short-term memory through an 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 close or further apart [39]. The first deep learning 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 predetermined. Sejnowski et al. [40] trained 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 over-fit. In addition, each token will be independently processed several times, in different time steps [39].

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) [41], 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. [42] noticed that even though SRNNs can learn short-term dependencies, long sequences led

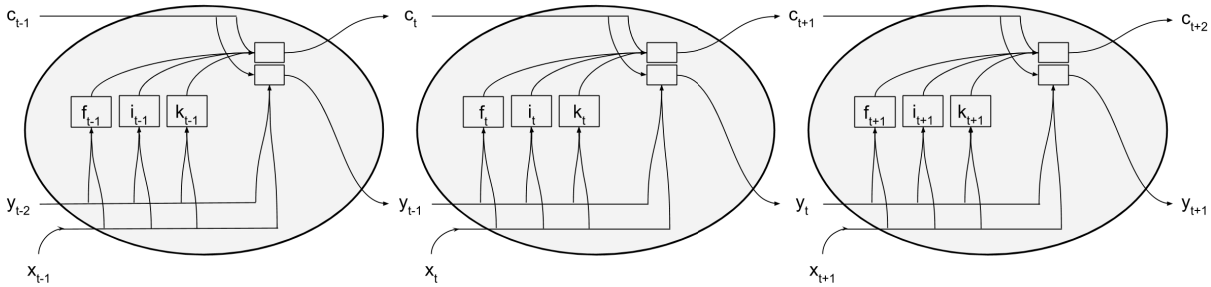


Fig. 1: Scheme of an LSTM, showing the units $t-1$, t and $t+1$.

to vanishing or exploding gradients, making it difficult for the model to learn them.

The LSTM unit [43] was developed to tackle this issue. LSTM models can maintain the error flow constant by introducing two gates – *input* and *forget* – that control how much information they let in [44]. 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 means everything passes). Equations (1) 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 the bias for the input gate, and U_f, W_f and b_f are the weight matrices and bias for the forget gate.

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

Fig. 1 shows how the different parts of an LSTM unit (represented by the grey circles) work. Each unit has a data flow c_t that carries information across time steps. In addition to the gates, there is also a simple hidden layer component, k_t, whose equation is given by Equation (2), in which U_k, W_k and b_k represent the weight matrices and bias for the hidden layer.

$$k_t = activation(dot(y_{t-1}, U_k) + dot(x_t, W_k) + b_k) \tag{2}$$

The next carry data flow c_{t+1} is computed by combining c_t, i_t, f_t and k_t, expressed by Equation (3):

$$c_{t+1} = i_t * k_t + c_t * f_t. \tag{3}$$

Finally, the output of the unit, which is also the state of the next unit, is calculated by Equation (4), where c_t is combined with the input x_t and the state y_{t-1}, via a dense transformation. Analogously to Equations (1) and (2), here we also have weight matrices U_y, W_y and V_y, and a bias vector b_y.

$$\begin{aligned} y_t = activation(&dot(y_{t-1}, U_y) + \\ &dot(x_t, W_y)) + dot(c_t, V_y) + b_y) \end{aligned} \tag{4}$$

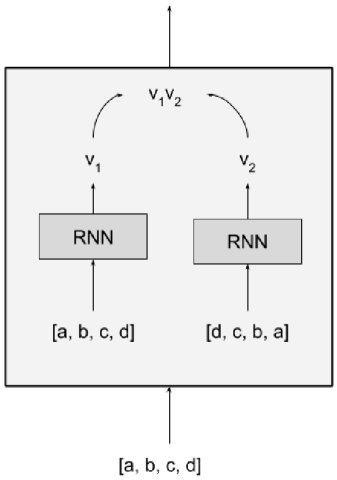


Fig. 2: Scheme of a BiRNN.

In the last few years, other recurrent units have appeared. The Gated Recurrent Unit (GRU) [45] 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 [44]. The Bidirectional Recurrent Neural Network (BiRNN) [46] can learn based not only in past but in future information as well, widening the context considered. Fig. 2 shows what happens inside a BiRNN layer. The BiRNN feeds into an RNN the input vector, and the reversed input vector to another RNN. In the end, the output is the combination of the outputs of both RNNs.

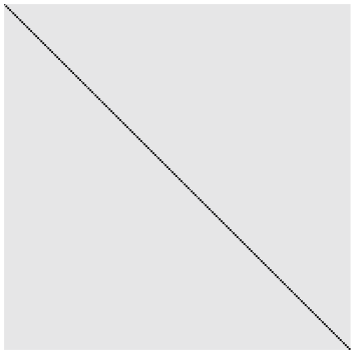
3) Input Representation: In NLP, it is necessary to represent text data as numeric vectors to be easily manipulated. The first thing to do is to decide the granularity of the representation: a vector may represent a sentence/sequence, a token (word or other character sequence separated by a space), a character, between others. For example, suppose it represents a token. There are two main representation methods: one-hot vectors and embedding vectors. One-hot vectors are binary vectors where only one entry has value 1. In one-hot representations, each feature corresponds to a token and

its value to whether the token is that feature's token. This results in a binary sparse vector with as many attributes as words in the vocabulary. This form of representation has been known to degrade the performance in neural network models [28]. 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 [28]. Also, using embedding vectors, we can choose the size and tune it to improve the model's performance.

The concatenation of these representations forms a matrix of parameters, which may be:

- *pre-trained*: there are specific models for embedding training (e.g., Word2vec [47] and GloVe [48]) 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;
- *fixed* or *dynamic*: when dynamic embeddings are used, we allow the matrix 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 fixed approach).

Fig. 3 shows two heat maps for the encodings of the opcodes of our data, one-hot on the left and embedding on the right, where each horizontal "line" represents one opcode. 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 over-fitting.



(a) One-hot encoding

(b) Embedding encoding

Fig. 3: Heat map for one-hot and embedding encodings of the opcodes.

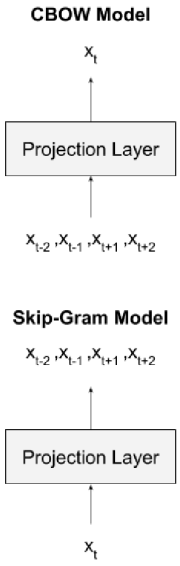


Fig. 4: High level view of both variants of the Word2vec model.

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 [49]. Normally, the second model achieves better results and it is the one generally used. In Fig. 4 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 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, Equation (5) shows the power of Word2vec in finding syntactic and semantic word relationships.

$$vec("Madrid") - vec("Spain") + vec("France") \simeq vec("Paris"), \tag{5}$$

III. Related Work

For some years now, ML has been used as a component in vulnerability detection models. It started as a support mechanism to automatize some parts of the task of finding vulnerabilities. For instance, Yamaguchi, Lindner and Rieck [50] 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 applies taint analysis to PHP code to extract possible vulnerabilities [14]. 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.

More recently, a few vulnerability detection models appeared that rely on ML for the classification task. Medeiros, Neves and Correia [51] extract code slices from PHP programs and translate them into an intermediate language developed by the authors. Finally, these translations are classified by a Hidden Markov Model to determine if they are vulnerable. A potential limitation of this approach is that the intermediate language used may simplify the code and loose insights about the computation behind it. Our "Assembly-like" intermediate language can decompose complex functions into simpler ones, exposing how they are executed and their interconnections inside the program. At the same time, it allows us to deal with a simpler language without the complex semantic of the high level program language.

Some works have used NLP techniques, such as Word2vec, to pre-train the embedding vectors [17], [52]. 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, as it happens in our case.

An interesting approach is suggested by Russel et al. [21], where the authors represent the data combining sentiment analysis with a CNN, and function level classification with an RNN. In the end, they classify the whole program using a Random Forest. Although it is an interesting work, they use a dataset of C/C++ programs, which are not prone to the same kind of software vulnerabilities as web applications.

In the related field of malware detection, Guo et al. [20] 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.

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 [17], [21], use methods that do not take into account the order of each token [52], or do not use an intermediate language easily scalable and flexible [51].

IV. Solution Proposal

We aim to classify PHP slices as SQLi vulnerable or non-vulnerable, by processing them in an intermediate language, composed of their respective opcodes. For that matter, we propose a DL model composed of Embedding, LSTM and Dense layers, following the guidelines presented in Section II-B. This is a novel approach that has not yet been used in the context of PHP vulnerability detection.

In this section, we will present the methodology used. We start by defining the dataset and the necessary preprocessing. Next, we introduce the network we are investigating, providing details for each layer, and the methods used to evaluate the model, including the metrics we considered.

A. PHP Programs Dataset

This section aims to present how we built our dataset, starting with a set of PHP programs and transforming them in their opcodes representation, generating thus a new dataset that will serve our DL model.

```php
<?php

$tainted = $_SESSION['UserData'];

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

$query = sprintf("SELECT * FROM '%s'", $t);

//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);

?>
```

(a) PHP code slice

line	#	op
3	0	FETCH_R
	1	FETCH_DIM_R
	2	ASSIGN
5	3	INIT_FCALL
	4	SEND_VAR
	5	SEND_VAL
	6	DO_ICALL
	7	JMPZ
6	8	ASSIGN
	9	JMP
8	10	ASSIGN
10	11	INIT_FCALL
	12	SEND_VAL
	13	SEND_VAR
	14	DO_ICALL
	15	ASSIGN
13	16	INIT_FCALL_BY_NAME
	17	SEND_VAL_EX
	18	SEND_VAL_EX
	19	SEND_VAL_EX
	20	DO_FCALL
	21	ASSIGN
14	22	INIT_FCALL_BY_NAME
	23	SEND_VAL_EX
	24	DO_FCALL
15	25	CONCAT
	26	CONCAT
	27	ECHO
17	28	INIT_FCALL_BY_NAME
	29	SEND_VAR_EX
	30	DO_FCALL
	31	ASSIGN
19	32	JMP
20	33	INIT_FCALL
	34	SEND_VAR
	35	DO_ICALL
22	36	ECHO
19	37	INIT_FCALL_BY_NAME
	38	SEND_VAR_EX
	39	DO_FCALL
	40	ASSIGN
	41	JMPNZ
23	42	INIT_FCALL_BY_NAME
	43	SEND_VAR_EX
	44	DO_FCALL
25	45	RETURN

(b) Opcode slice obtained with the VLD tool

[82, 83, 40, 63, 119, 67, 131, 45, 40, 44, 40, 63, 67, 119, 131, 40, 61, 118, 118, 118, 62, 40, 61, 118, 62, 10, 10, 42, 61, 68, 62, 40, 44, 63, 119, 131, 42, 61, 68, 62, 40, 46, 61, 68, 62, 64]

(c) Numeric vector

Fig. 5: Example of a code slice and the successive transformations it suffers.

TABLE I: Vocabulary composed of VLD opcodes. The index of the opcode in position i, j is given by $i * 5 + j + 1$.

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

1) Raw Dataset: Stivalet and Fong [53] developed a PHP test case generation tool[2] to generate the SARD test cases. Each sample is composed of a comment section labeling it, and a code slice. The code slice starts with an entry point and ends in a sensitive sink.

From SARD, we use as raw data 1362 SQLi test cases, namely 858 vulnerable and 504 non-vulnerable. Non-vulnerable instances are code slices where the user input is correctly sanitized or validated. On the other hand, vulnerable slices do not sanitize or validate the input correctly. Note that 1) the input may be sanitized and still compromise the application, and 2) the malicious input may be propagated across the slice throughout the assignments to other variables. These show the complexity of the task we intend to solve.

2) Data Transformation and Processing: Traditionally, PHP uses a virtual machine engine called Zend [3] to compile and run the programs. Zend transforms the PHP programs into opcodes, which are then executed. A tool called Vulcan Logic Dumper (VLD)[4] is able to intercept the opcodes processing before they are executed, allowing them to be saved into a file. This way, we gain access to an intermediate language representation in which the original functions have been partitioned into simpler ones and have a more restricted space. Another advantage of analyzing programs in an intermediate language is that it may be used for more than a high-level programming language, turning the model fit for any programming language that may be represented by this intermediate language.

We started by executing all examples on VLD, obtaining a PHP opcode slice for each PHP slice. We also defined the vocabulary composed of all valid opcodes used in VLD, which will be useful when training the model. Table I lists this vocabulary. Besides the PHP opcodes, we added a special token to designate out of vocabulary tokens that may occur, the *OOV* token. The index of the opcode in position (i, j) (row, column) is given by expression $i * 5 + j + 1$.

Since neural networks receive as input arrays of numerical values, we created a numeric vector for each instance, by mapping the opcodes to their corresponding index in the vocabulary. Fig. 5 shows an example of how the numeric vector (part (c)) is obtained from a code slice (part (a)) of our dataset, which is transformed in the opcode slice by the VLD tool (part (b)). As we can observe the first opcode of the slice, *FETCH_R*, has index 82 (see Table I by applying the expression $16 \times 5 + 1 + 1 = 82$), which is the first element of the vector.

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.

B. Model

Fig. 6 gives a high-level view of the model we propose. The model 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. This way, the input vectors go sequentially through the Embedding, LSTM, Dropout, and Dense layers, suffering successive transformations and producing the final output.

The LSTM + Dropout layers can be stacked n times to increase the learning capacity of the model. Table II lists

[2]The tool is available in SARD

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

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

TABLE II: 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

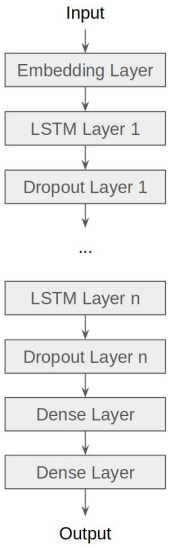


Fig. 6: High level view of our model.

each layer's input and output size, 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*), but the subsequent LSTM layers simply transform vectors into other vectors. *MAX_LENGTH* corresponds to the maximum sequence size we allow and $\textit{HIDDEN\_SIZE}_i$ is a predefined value that needs to be tuned along with n, where $i \in \{1, \ldots, n\}$ and $n \in \mathbb{N}$. The choice of activation function of each layer follows the recommendations of the Keras documentation [54] for the LSTM layer and of Goodfellow, Bengio and Courville [24] for the Dense layers. They state that the *ReLU* is the preferred activation function for neural network layers and *Sigmoid* for output layers in classification problems with two classes. Next, we present each layer in detail.

1) Embedding Layer: The Embedding Layer maps tokens to embedding vectors. We will investigate three embedding approaches:

- *dynamic*: the matrix is initialized from a uniform distribution [54] and its values are updated according to the backpropagation algorithm and the training data [44];
- *pre-trained* and *fixed*: the embedding vectors for each opcode are pre-trained using the word2vec skip-gram model. Next, the vectors are used to initialize the weight matrix. During training, the values will not change;
- *pre-trained* and *dynamic*: it follows the same idea as the previous point but now we allow the values to change during training, to adapt them to our task's data.

Note that for the pre-trained approaches, we must build a corpus from PHP applications that will be used as training data for the Word2vec model.

2) LSTM Layer: After learning the embedding vectors for each opcode, we want to learn patterns present in the opcode slice. In the vulnerability detection task, the order of the opcodes in the slice is very relevant. For that matter, we chose to stack n LSTM layers, since they can produce output vectors that encode information of previous opcodes and their order. We started off with a single layer and then studied the impact of adding additional layers, following Chollet's recommendations on how to balance under and over-fitting in neural networks [44].

3) Dropout Layer: During training, this layer randomly sets some entries of its input to zero, according to a given probability (δ). The goal of this approach is to introduce noise in the model to prevent it from memorizing irrelevant patterns that are learned by chance by the LSTM layer. At test stage, the layer does not apply any transformation to its input [44]. Although it takes more time for the model to converge, neural network models that have dropout layers can reduce over-fitting further and improve their performance [55]. We decided to add a Dropout layer after each LSTM layer of the model.

4) Dense Layers: The last two layers are fully-connected feed-forward neural network layers. The first Dense Layer's role is to learn the relationship between the slice and the corresponding label (vulnerable or non-vulnerable). It transforms the input in a vector with the same shape that grasps this relationship. The last layer classifies the slice. It follows a common configuration in classification problems, such as the one we are solving, by transforming the input in a numeric value between 0 and 1 through the *Sigmoid* activation function [44]. In this context, the value represents the probability of having an SQLi vulnerability in that code slice.

C. Evaluation

To evaluate the model, we applied a 70/30 random stratified train-test split to the dataset, which maintains the percentage of vulnerable/non-vulnerable samples in each set. The training set is therefore composed of 953 samples and the test set of 409, where 63% are vulnerable and the rest non-vulnerable. 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, based on the rate of correctly classified examples (known as *accuracy*), the true positive rate (or *recall*), and the rate of true positives given all predicted positives (known as *precision*). Our goal is then to better balance these 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, that could easily be 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 is usually higher.

V. Experimental Results

In this section, we will dive into the experiments performed for the 1 LSTM layer model. We will begin by giving details on the implementation. Next, we describe the methods and decisions taken for the experiments, as well as the various configurations tested in order to find the most suitable for our task. We finally present the results obtained.

A. Implementation

We investigated the model with one block of LSTM + Dropout layers. For that, we used the well-known Python package Keras [54] to implement the various model configurations. Keras provides a convenient and easy-to-use interface for developing neural networks. It works as a wrapper for the opensource Tensorflow package [56], which needs more details to configure a model.

B. Configuration of the Experiments

Throughout the experiments, we fixed the seed for the train-test split (depicted in Section IV-C) so that all models were trained and tested with the same data. In addition to helping reproducibility, we believe it makes the comparisons between configurations fairer.

For our experiments, we decided to test the ADADELTA, RMSProp, and ADAM optimizers to determine which is the most suitable for our problem. Neural networks have a considerable amount of hyperparameters. Hence, to simplify the first experiments and gain some intuition on the problem, we decided to leave the optimizers' hyperparameters to their default values. Table III lists these values.

Regarding other hyperparameters, we chose to tune the number of units in the LSTM layer, δ, and the number of epochs, evaluating them in different configurations. Next, we detail each hyperparameter:

- Number of units in the LSTM layer, that corresponds to the *HIDDEN_SIZE*$_1$. Let us denominate it simply by *HIDDEN_SIZE*, since there is only one layer and only one *HIDDEN_SIZE*$_i$ to tune. This hyperparameter is strongly related to the model's learning capacity: the more units the layer has the more it learns. However, the trade-off is that it may start to have too many parameters to learn and over-fit, loosing generalization power;
- δ, which represents the dropout rate, a value between 0 and 1 associated to the Dropout Layer, and corresponds to the probability of each entry of the input vector turning into 0;
- Number of epochs, which defines the number of times the optimizer updates the parameters. Generally, the higher this parameter is the better the model learns. But again, we need to balance learning with generalization power and not let the model over-fit.

We performed manual hyperparameter tuning for these three parameters, testing one by one, and following the order they appear in the list above. Table IV shows the configurations tested for each hyperparameter and optimizer combination. For each hyperparameter, we started by testing a range of sparser values: $\{10, 20, 40, 80, 160\}$ for the *HIDDEN_SIZE* and the number of epochs, and $\{0.2, 0.3, 0.5\}$ for δ. Next, we fine-tuned the search testing two values around the best result (one smaller and one greater), repeating the process as many times as necessary. Each configuration was tested for 10 epochs (except, of course, when tuning the number of epochs).

TABLE III: 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

TABLE IV: 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

C. Results

According to our evaluation approach, described in the previous section, the results of each optimizer, trained with the best values found for each hyperparameter are shown in Table

TABLE V: 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

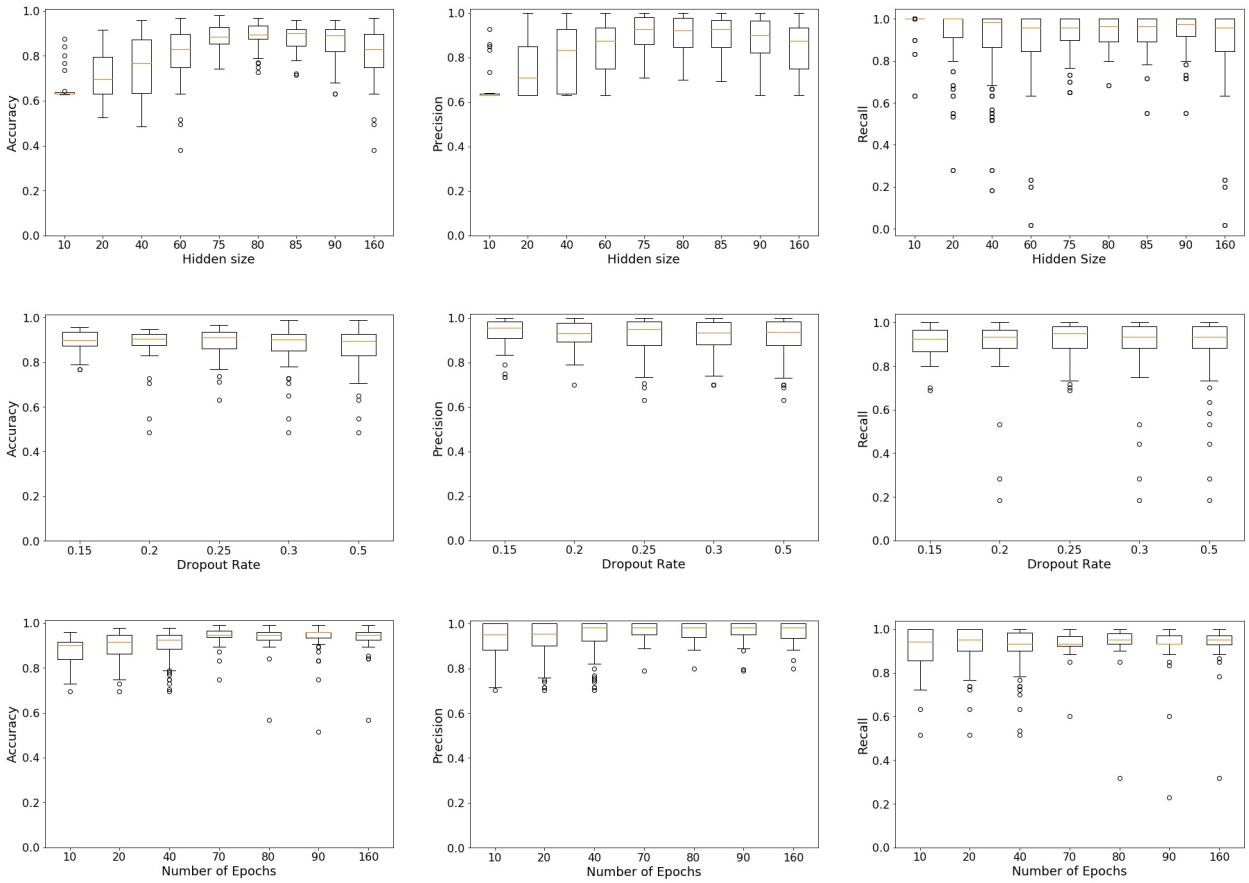


Fig. 7: Box-plots for the hyperparameter tuning of the hidden size of the LSTM layer, dropout rate and number of epochs (respectively, the first, second and third rows), using the RMSProp optimizer.

V. As we can observe the worst optimizer is ADADELTA since its performance is the worst for all metrics. Furthermore, it is the optimizer that takes the longest to converge - it takes 160 epochs, against 70 epochs for RMSProp and 35 for ADAM. ADAM and RMSProp have good results in different metrics: RMSProp has the best accuracy and recall, and ADAM has the best precision. Note that ADAM can achieve nearly as good accuracy as RMSProp (0.9413 against 0.9535) with half the epochs (35 against 70) and less units (70 against 80). For our work, it is more important the metrics' results than less computational cost, so we consider that RMSProp achieves the best results under these hypotheses.

Fig. 7 shows 9 box-plots that result from the tuning for the 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 box-plots tend to improve as the value increases, until it reaches a point where the performance starts to degrade. There are some aspects we should consider when analysing multiple box-pots and multiple metrics: (1) we always want to balance the performance for the three metrics, though the accuracy is slightly more important; (2) it is important to look not only to the median of each plot but to the variance, represented by the height of the box; and (3) finally, between two similar box-plots where one has lower outliers, we should choose the other one. Based on these guidelines, we chose the values 80 for the number of hidden units, 0.15 for the dropout rate, and 70 for the number of epochs.

VI. Conclusions

This paper aims to advance in the state-of-the-art of vulnerability detection in web applications, with a focus on SQLi vulnerabilities. For that, we propose a Deep Learning (DL) model for Natural Language Processing (NLP) that processes PHP code translated into an intermediate language. The model is constituted of different layers, namely Embedding, LSTM + Dropout, and Dense layers. Also, the paper presented the evaluation of the model for one LSTM with three optimizers (ADADELTA, RMSProp, and ADAM) and different configurations for their hyperparameters. RMSProp achieved the best scores. The experiments so far have validated our approach, with our best model achieving results above 95% for the accuracy, precision and recall. Nonetheless, the train and test sets are retrieved from the same generated dataset,

which might not represent fully real PHP applications. In the future, this work should be tested with a dataset of real PHP applications to minimize this issue. Nevertheless, the preliminary results lead us to conclude that the proposed DL model can discover SQLi vulnerabilities, thus taking a further step towards the detection of vulnerabilities based on machine learning.

ACKNOWLEDGMENT

This work was partially supported by the ITEA3 European through the XIVT project (I3C4-17039/FEDER-039238), and national funds through FCT with reference to SEAL project (PTDC/CCI-INF/29058/2017), and LASIGE Research Unit (UIDB/50021/2020).

REFERENCES

[1] CVE, "CVE Details. The ultimate security datasource," https://www.cvedetails.com/browse-by-date.php.

[2] 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*, vol. 29, 2017.

[3] DarkReading, "Sql injection attacks represent two-third of all web app attacks," 2019, https://www.darkreading.com/attacks-breaches/sql-injection-attacks-represent-two-third-of-all-web-app-attacks/d/d-id/1334960.

[4] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking static analysis tools for web security," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1159–1175, Sept 2018.

[5] I. Medeiros, N. F. Neves, and M. Correia, "Equipping WAP with weapons to detect vulnerabilities," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[6] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, Jun. 2006, pp. 27–36.

[7] P. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2015.

[8] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

[9] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the International World Wide Web Conference*, Apr. 2014, pp. 63–74.

[10] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[11] P. Godefroid, M. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," pp. 1–20, 2012.

[12] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 2014, pp. 37–48.

[13] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[14] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2015.

[15] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1293–1296.

[16] I. Medeiros, N. F. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Jul. 2016.

[17] 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," in *Annual Network and Distributed System Security Symposium*, Feb. 2018.

[18] 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 *Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems*, Dec. 2019, pp. 10 197–10 207.

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

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

[21] 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*, 2018, pp. 757–762.

[22] WhiteHat Security, "The DevSecOps Approach - Using AppSec Statistics to Drive Better Outcomes," Nov. 2019.

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

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

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

[26] 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*, vol. 4, no. 2, pp. 26–31, 2012.

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

[28] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 10, no. 1, pp. 1–309, 2017.

[29] 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*, vol. 2, 2010, pp. 1045–1048.

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

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

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

[33] 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.

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

[35] 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*, 2017, pp. 5998–6008.

[36] 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.

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

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

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

[40] T. J. Sejnowski and C. R. R., "Parallel networks that learn to pronounce english text," *Complex Systems*, vol. 1, no. 1, pp. 145–168, 1987.

[41] J. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[42] Y. Bengio, P. Simard, P. Frasconi *et al.*, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.
[43] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[44] F. Chollet, *Deep Learning with Python*. Manning Publications Company, 2017.
[45] 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*, pp. 1724–1734, 2014.
[46] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
[47] 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*, 2013, pp. 3111–3119.
[48] O. Levy, Y. Goldberg, and I. Dagan, "Improving distributional similarity with lessons learned from word embeddings," *Transactions of the Association for Computational Linguistics*, vol. 3, pp. 211–225, 2015.
[49] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
[50] 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*, 2011, pp. 13–13.
[51] I. Medeiros, N. Neves, and M. Correia, "Statically detecting vulnerabilities by processing programming languages as natural languages," *arXiv preprint arXiv:1910.06826*, 2019.
[52] 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*. ACM, 2016, pp. 85–96.
[53] 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*, 2016, pp. 409–415.
[54] F. Chollet *et al.*, "Keras," https://keras.io, 2015.
[55] 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*, vol. 15, no. 1, pp. 1929–1958, 2014.
[56] T. G. Brain, "Tensorflow," https://www.tensorflow.org/, 2015.