Statically Identifying XSS using Deep Learning

Heloise Maurel, Santiago Vidal, Tamara Rezk

Statically Identifying XSS using Deep Learning

Heloise Maurel[1], Santiago Vidal[2][a] and Tamara Rezk[1][b]
[1]*INRIA, INDES Project, Sophia Antipolis, France*
[2]*ISISTAN-CONICET, Argentina*
{*heloise.maurel, tamara.rezk*}*@inria.fr, santiago.vidal@isistan.unicen.edu.ar*

Keywords: Web security, Deep learning, Web attacks, Cross-Site Scripting.

Abstract: Cross-site Scripting (XSS) is ranked first in the top 25 Most Dangerous Software Weaknesses (2020) of Common Weakness Enumeration (CWE) and places this vulnerability as the most dangerous among programming errors. In this work, we explore static approaches to detect XSS vulnerabilities using neural networks. We compare two different code representations based on Natural Language Processing (NLP) and Programming Language Processing (PLP) and experiment with models based on different neural network architectures for static analysis detection in PHP and Node.js. We train and evaluate the models using synthetic databases. Using the generated PHP and Node.js databases, we compare our results with a well-known static analyzer for PHP code, ProgPilot, and a known scanner for Node.js, AppScan static mode. Our analyzers using neural networks overcome the results of existing tools in all cases.

1 INTRODUCTION

Cross-Site Scripting, a.k.a. XSS, vulnerabilities are injection flaws (Fogie, 2007) in web applications caused by untrusted inputs flowing to sensitive parts of the application during its execution. Untrusted inputs are commonly called sources and sensitive targets are called sinks. The principle to programmatically prevent an XSS vulnerability is simple and boils down to the use of correct sanitizers: absence of a correct sanitizer (Hooimeijer et al., 2011) in a path from source to sink directly implies a potential vulnerability. XSS vulnerabilities have been largely studied in the literature (e.g., (Balzarotti et al., 2008), (Gundy and Chen, 2009), (Luo et al., 2011), (Lekies et al., 2017), (Melicher et al., 2018)) and are relatively well understood by now. For example, the OWASP XSS cheatsheet series project[1] proposes a series of nine rules to correctly sanitize or avoid untrusted inputs in different contexts to prevent XSS attacks. Thus, in theory, the XSS problem is solved. In practice, inserting the correct sanitizer in the correct context is tricky, and XSS is still listed as one of the most common vulnerabilities of web applications today (see, e.g., OWASP's Top Ten Project[2]).

Different approaches can be used to detect and prevent XSS vulnerabilities. One approach is to use black-box vulnerability scanners (Doupé et al., 2010) that test different crafted untrusted inputs to exploit a vulnerability in the application. If an attack is found, the conclusion is clear: a vulnerability exists. Black-box vulnerability scanners have as their main advantage their independence of the web application's technology and their capacity to give a proof of concept of a concrete exploit if a vulnerability exists. Other approaches use static and dynamic or hybrid analyses to detect XSS vulnerabilities. A well-known analysis to detect XSS is taint tracking[3,4] (Schoepe et al., 2016), which consists of tracking flows from sources to sinks in the application. Taint tracking analyses have the advantage of precisely determining if a path from source to sink in the application contains a potential vulnerability.

On one side of the spectrum, dynamic analysis (e.g., dynamic taint tracking analyses (Melicher et al., 2018)) is usually easier to implement but holds the disadvantage of analyzing paths that are dynamically explored, that is, executed on specific inputs. On the opposite side of the spectrum, static analysis can cover all paths from sources to sinks and requires

[a] https://orcid.org/0000-0003-2440-3034
[b] https://orcid.org/0000-0003-3744-0248
[1] https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[2] https://owasp.org/www-project-top-ten/
[3] http://perldoc.perl.org/perlsec.html
[4] http://phrogz.net/programmingruby/taint.html

source code. Common static approaches to prevent XSS are based on taint tracking (Livshits and Chong, 2013) or types[5] (Chlipala, 2016), to name a few. The most significant disadvantage of XSS static detectors is that they require a considerable design and implementation effort, and they are entirely dependent on the web application's technology. Moreover, they often need to deal with the impedance mismatch problem due to the mix of languages found in the backend and front-end of web applications (Samuel et al., 2011; Saxena et al., 2011).

In this work, we explore static approaches to detect XSS vulnerabilities using neural networks. Our thesis is that, compared to other static approaches, neural networks can provide us with precise XSS analyzers by adapting to different web technologies in a reasonably straightforward manner.

In order to prove this thesis, we compare and tweak for the XSS detection problem two different code representation techniques, based on natural language processing (Allamanis et al., 2016) (NLP) and programming language processing (Allamanis et al., 2018; Alon et al., 2019) (PLP), respectively, and use deep learning[6] for the detection of XSS vulnerabilities in code written in two mainstream web server-side languages: PHP[7] and Node.js[8]. Machine learning techniques have been previously applied to the detection of security vulnerabilities (e.g., (Li et al., 2018; Staicu et al., 2018; Calzavara et al., 2019; She et al., 2020)). . However, to the best of our knowledge, no previous work has attempted to investigate different code representations to statically detect XSS vulnerabilities in different languages.

One major challenge we encounter to move forward towards this goal is to find datasets of server-code sources that are reliably classified as XSS vulnerable (unsafe) or XSS free (safe). For XSS vulnerabilities of the third kind, a.k.a. DOM XSS, it is enough to analyze the front-end code of the web application for static detection. Front-end code can easily be obtained[9] by crawling the web, but it is not enough to prove our thesis since we aim at comparing results in different languages, not only JavaScript embedded in HTML. To address the challenge of obtaining appropriate datasets, we started by using the PHP-based dataset provided by NIST in the SAMATE project[10]. We found inconsistencies[11] between the PHP dataset generator and the sanitizing rules of OWASP XSS cheatsheets[1] that led to attacks in several samples classified as safe (see Fig. 2 as an example). Hence, we modified the code generator used by NIST to obtain a more reliable and larger dataset (see Table 3). Using the modified PHP generator, consistent with the rules of OWASP XSS cheatsheets, we also obtained a Node.js generator to obtain a second dataset. Our results show that PLP representations, which encode more knowledge about the semantics of the language into the model, are better suited than NLP representations in both cases, PHP and Node.js. In the case of PHP, we also perform experiments for NLP to evaluate the analyzers' results when only using PHP and when including HTML/JavaScript code. Our results show that by analyzing PHP code along with HTML code, the results are better than only analyzing PHP code.

We have further compared our results to well-known static analysis tools such as ProgPilot[12] for PHP and AppScan[13] for Node.js, finding that our results overcome the results of these tools in all cases.

Contributions. In summary, our contributions are:

- We compare two different code representations based on NLP and PLP and generate models using different neural network architectures for static analysis detection in PHP and Node.js.
- We rebuild the PHP generator offered by NIST to correct inconsistencies w.r.t. the OWASP cheatsheets rules for XSS and enlarge the size of the PHP dataset. We build a new generator for Node.js code. We evaluate models in two different datasets of PHP code: one including HTML/-JavaScript as code and one including it as text.
- Using the generated PHP and Node.js datasets, we compare our results with a well-known static analyzer for PHP code, ProgPilot, and a known scanner for Node.js, AppScan static mode. Our analyzers using neural networks overcome the results of existing tools in all cases.

[5]https://w3c.github.io/webappsec-trusted-types/dist/spec/

[6]We choose to use deep learning rather than classical machine learning approaches because classical machine learning approaches require the manual discovery of model features that turns out to be a complex problem since we want to deal with different web technologies.

[7]http://www.php.net

[8]http://nodejs.org/en/

[9]Notice that frontend code is easy to obtained but still not easy to classify (Melicher et al., 2018).

[10]https://samate.nist.gov/Main_Page.html

[11]We have communicated these problems to NIST in November 2020. They acknowledged and offer our modifications to the generator to solve the inconsistencies.

[12]https://github.com/designsecurity/progpilot

[13]https://www.hcltechsw.com/products/appscan

2 DATASETS FOR XSS

One of the essential steps to apply any supervised deep learning algorithm is to design a reliable and comprehensive dataset. In our case, the server-side code cannot be obtained by browsing the web, and it is difficult to reliably and automatically classify the server-side code on public repositories, like XSS-safe or unsafe. Thus, we explore the use of a synthetic generator of vulnerabilities for XSS flaws.

2.1 PHP Dataset at NIST

We used the PHP generator provided by the SAMATE project at the National Institute of Standards and Technology (NIST) [14]. The generator can generate different synthetic PHP source codes affected for a given vulnerability. For XSS, it generates 10,080 samples distributed as follows:

- 5,728 safe samples (57%)
- 4,352 unsafe samples (43%)

To generate these examples, the PHP generator combined all the possibilities between 21 HTML construction templates that follow the OWASP XSS rules[15], 16 PHP user input templates, and 30 diverse types of proper/improper sanitizations to mitigate XSS.

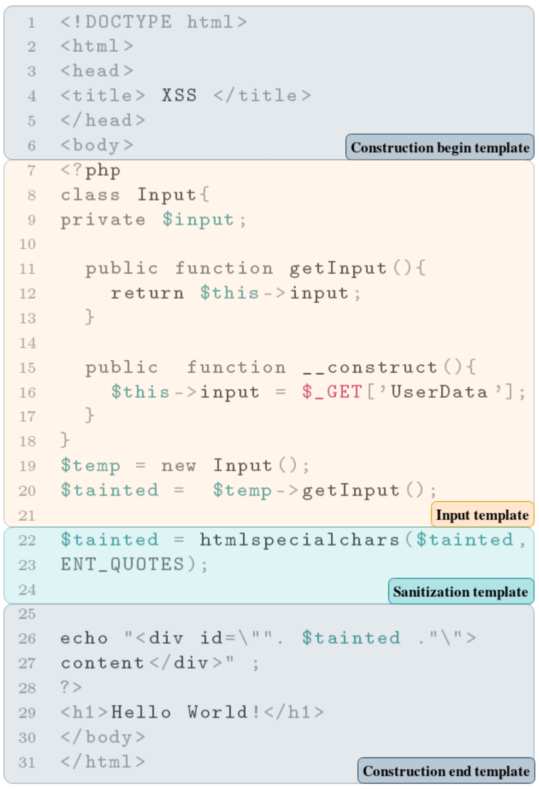
$\#generatedSample = \#construction * \#input * \#sanitization$

Figure 1 illustrates how each generated sample is a combination of these components:

- The first and the last fragments are construction templates that present one of the possibilities to begin and end a PHP sample. Each ending part is the implementation of one OWASP rule. In the example, the inside double quote attribute template of rule #2 is used.
- The second fragment is an input template composed of one of the distinctive user inputs in PHP. In the example the $_GET[] primitive array is used.
- The third fragment is a sanitization template composed of one option to properly/improperly sanitize or not sanitize a PHP sample. PHP provides several kinds of built-in functions like the htmlspecialchars function that is used in this example.

[14]https://samate.nist.gov/Main_Page.html

[15]The OWASP Rules are a series of XSS Prevention Sheets covering the vast majority of typical use cases to clean up sources in several contexts. Each rule covers distinctive types of sink templates and gives the appropriate sanitization to clean them up.



```
<!DOCTYPE html>
<html>
<head>
<title> XSS </title>
</head>
<body>
<?php
class Input{
private $input;

  public function getInput(){
    return $this->input;
  }

  public  function __construct(){
    $this->input = $_GET['UserData'];
  }
}
$temp = new Input();
$tainted =  $temp->getInput();

$tainted = htmlspecialchars($tainted,
ENT_QUOTES);


echo "<div id=\"". $tainted ."\">
content</div>" ;
?>
<h1>Hello World!</h1>
</body>
</html>
```

Figure 1: Three template elements to generate a single sample.

2.2 Issues Found in the NIST Database

We found 95 misclassifications in the original PHP generator:

- 74 samples misclassified as safe instead of unsafe due to:
  - specific PHP type juggling errors.
  - mismatching between PHP sanitization function capabilities and the properly sanitization recommendations given by the OWASP rules. An example of such misclassification is shown in the exploit[16] presented in Figure 2. In this example, the addslashes function is used in the sanitization template (line 23). However, this function cannot sanitize rules 1, 2, 3, 4, and 5 because open-angle brackets (<) can break out the double-quoted attribute.
- Twenty-one samples misclassified as unsafe instead of safe, especially samples with no flow between the source and the sink of the potential XSS vulnerability.

[16]the original sample can be found at http://samate.nist.gov/SARD/view_testcase.php?tID=191377

2.3 Fixing and Extending the Generator

```
<!-- Begin construction template -->
<!DOCTYPE html>
<html>
<head>
<title> XSS </title>
</head>
<body>
<?php
//Input template
class Input{
  private $input;
  public function getInput(){
    return $this->input;
  }
  public  function __construct(){
    $this->input = $_GET['UserData'];
  }
}
$temp = new Input();
$tainted =  $temp->getInput();

//sanitization template
$tainted = addslashes($tainted);

//End construction template
echo "<div id=\"". $tainted ."\">content
</div>";
?>
<h1>Hello World!</h1>
</body>
</html>
```

Figure 2: Example of improper sanitization template with addslashes function classified as SAFE by NIST. An exploit to launch the XSS attack is : ><script>alert(1)</ script > used in the URL parameter UserData.

To fix the 95 classification errors, we corrected, added, and combined predicate attributes describing each sanitization template (e.g., properly htmlspecialchars-ENT_QUOTES sanitization function in Figure 1 or improperly addslashes sanitization function in Figure 2) according to the OWASP rules recommendations to sanitize the HTML templates safely[17].

We also extended the PHP generator (Table 1) with 25 sink templates, 16 XSS inputs, and 58 different kinds of proper/improper sanitizations. As a result, we could generate 23,200 different samples of XSS safe and unsafe PHP code.

One advantage of using this kind of generator is that it can create a reliable database for different programming languages without changing the implementation. Thus, to generate a dataset for Node.js, we create 121 code templates that allow us to obtain more than 25,000 samples of safe and unsafe Node.js files (Table 1).

Table 1: Vulnerability generator characteristics

Language	Template	# of template	# of generated samples
PHP	Input	16	23,200
	Construction	25	
	Sanitization	58	
Node.js	Input	12	25,200
	Construction	25	
	Sanitization	84	

2.4 Final Datasets for PHP and Node.js

Let us assume that D0 represents the final PHP database after the fix and the extension described in Table 1. We created 2 transformations, namely:

1. rename_randomly_variable_names() that takes D0 and rename all the variable names randomly to obtain D1.
2. echo_html() that takes the database D1 and put all the HTML templates on a PHP echo to obtain D2.

In this way, we created two PHP databases (Table 3):

- D1 - composed of D0 and rename_randomly_variable_names(D0) which has a total of 23,200 samples averaging 23 lines of code (LOC) each sample.
- D2 - composed of echo_html(D1) with a total of 23,200 samples (24 LOC on average).

In the case of Node.js, we only used D2 because the front-end code was put in the write JavaScript function, as it is custom for Node.js code in practice. The D2 dataset for Node.js has a total of 24,566 samples (36.7 LOC on average).

3 XSS IDENTIFICATION

To identify XSS in the source code through deep learning, we experimented with two approaches [18]. First, we designed an approach that represents the source code based on the tokenization of its contents. Second, we adapted the Code2Vec deep learning network (Alon et al., 2019) that represents the source code based on the AST. In the following, we describe both approaches.

[17]Our generators and datasets can be found at https://gitlab.inria.fr/deep-learning-applied-on-web-and-iot-security/statically-identifying-xss-using-deep-learning

[18]The implementation of both approaches can be found at https://gitlab.inria.fr/deep-learning-applied-on-web-and-iot-security/statically-identifying-xss-using-deep-learning

Table 2: Summary of OWASP rules used by the PHP and NodeJS generators in the construction HTML templates
** symbol indicate the constructions that we added compared to the initial generator and the official OWASP rules*

#rule	Summary	Context template code	#template
0	Never insert untrusted data except in allowed locations	<**script**>$tainted</**script**> <!−−*$tainted*−−> <**div** $tainted=test /> <$tainted **href=**"/ test " /> <**style**>$tainted</**style**> <**body onload=**"xss()">*	6
1	HTML encode before inserting untrusted data into HTML element content	<**body**>$tainted</**body**> <**div**>$tainted</**div**>	2
2	Attribute encode before inserting untrusted data into HTML common attributes	<**div** attr= $tainted >content</**div**> <**div** attr=' $tainted '>content</**div**> <**div** attr=" $tainted ">content</**div**>	3
3	Javascript encode before inserting untrusted data into javascript data values	<**script**>alert(' $tainted ')</**script**> <**script**>alert(\"". $tainted ."\")</**script**>* <**script**>x='$tainted'</**script**> <**script**>x="$tainted"</**script**>* <**div onmouseover=**"x='$tainted'"</**div**> <**div onmouseover=**'x="$tainted"'</**div**>* <**script**>window. setInterval (' $tainted ') ;</ **script**>	7
4	CSS encode and strictly validate before inserting untrusted data into HTML style property values	<**style**>selector{property : $tainted ;}</**style**> <**style**>selector{property :" $tainted ";}</**style**> <**style**>selector{property :' $tainted '";}</ style > <**span style=**"property : $tainted ">text</**span**>	4
5*	URL encode before inserting untrusteddata into HTML URL parameter values	<**a href=**"$tainted">link</**a**> <**a href=**$tainted>link</**a**>* <**a href=**' $tainted '>link</**a**>*	3

3.1 Concatenation Technique (NLP)

This representation used an existing neural network approach for words used in natural language processing (NLP) called Word2Vec (Mikolov et al., 2013). Word2Vec presents a group of models that can create and train semantic vector spaces (in general of several hundred dimensions) based on a text corpus. In this vector space, each word in the corpus is represented as a vector (called embedding). Each vector is positioned in the space so that words sharing similar contexts are closely located. In our case, we use source code as a corpus being the "words" the different tokens in the source code. In this way, we kept the Node.js and PHP source code meaning by keeping the context between the words that compose the code.

As it is shown in Figure 3, we start by listing the different tokens in the dataset. Each token represents a word on a target source code, such as variable names, keywords of the programming language, etc. Similarly, each line of a target source code is represented by a token list. Word2Vec is applied to create a dictionary of word embedding using the token list where the keys are the tokens of the complete source code and where a unique vector represents each key.

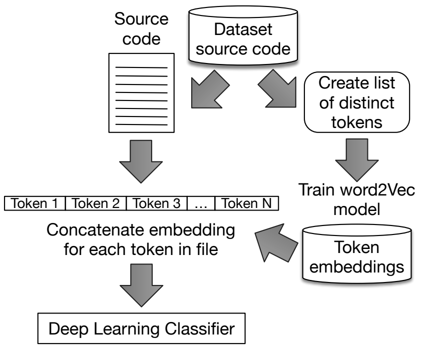

Source code
Dataset source code
Create list of distinct tokens
Train word2Vec model
Token embeddings
Token 1 | Token 2 | Token 3 | ... | Token N
Concatenate embedding for each token in file
Deep Learning Classifier

Figure 3: Word-based representation approach.

At this point, we have the representation of words from the source code, but we do not have a vectorial representation of them. To create a vectorial representation of a given source code file, we concatenated each of the vectors associated with the tokens that appear in a single file, as shown in Figure 3. Thus, each file is represented as the concatenation of the Word2Vec embeddings of the tokens present in the file. These concatenated embeddings will be the input of the deep learning classifier; details of this classifier can be found in Section 4.1.

Table 3: Generated Databases

Language	Database	Classification			Mismatched distributions			
		Total	#safe	#unsafe	Set	#rule	#safe	#unsafe
PHP	D1 HTML	23,200	12,544	10,656	train	3,4	5,152	5,056
					test	0,1,2,5	3,657	2,839
					validation		3,735	2,761
	D2 echo HTML	23,200	12,544	10,656	train	3,4	5,152	5,056
					test	0,1,2,5	3,657	2,839
					validation		3,735	2,761
Node.js	D2 write HTML	24,566	13,164	11,400	train	3,4	5,736	5,052
					test	0,1,2,5	3,703	3,185
					validation		3,725	3,163

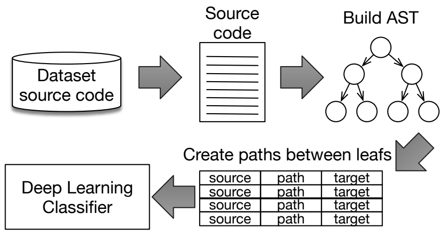


Figure 4: Hashed-AST based representation approach.

3.2 Hashed-AST Technique (PLP)

The second approach is based on the AST representation proposed by Code2Vec. Code2Vec proposes a technique to represent embeddings to be used in Deep Learning models that seeks to analyze source code. In this way, Code2Vec proposes to represent a piece of code by extracting information from its AST. Specifically, given the AST of a piece of code, Code2Vec proposes registering each possible path between leaves as a "path context". A path context is a triplet $< source, path, target >$ that registers the initial leave (source), the destination leave (target), and the path between them. Each piece of code is represented by a bag of path contexts. An attention mechanism is used to learn the importance of each path context to make a prediction. In this context, our approach creates an AST for each source code file (Figure 4). Then, all the possible paths between leaves are created and represented by a triplet $< source, path, target >$. Thus, each file is represented by a bag of path contexts. To control the number of possible triplets used to represent a file, three parameters are used:

- *maxContext:* limits the maximum number of triplets used to represent a file.
- *maxPath length:* binds the highest number of nodes in the tree that is need to traverse before founding a common node when creating a path between 2 leaves.
- *maxPath width:* restricts the maximum number of leaves between 2 leaves when creating a path.

Since the Code2Vec implementation only supported Java and C# source code at the time of experimenting, we implemented the AST representation for JavaScript and PHP source code using open source parsers. As far as we know, there is not a parser that creates a combined AST of two or more languages (e.g., an AST of PHP and HTML). For this reason, differently from the previous approach, this one can only analyze the database D2 for PHP.

Regarding the deep learning classifier used in this approach, we used the neural network based on attention originally proposed by Code2Vec (Alon et al., 2019). Since we focus on a binary classification problem, we modified the final softmax layer for a sigmoid function. On the same line, we also modified the input representation of the targets present in the triplets. In short, the classifier starts by creating an embedding for each triplet ($< source, path, target >$). Then, a dense layer learns how to combine the embeddings for a given file. Using these combined embeddings, an attention layer is calculated. The output of the attention layer is used to predict if the file is safe or unsafe using a sigmoid function (most details of this classifier can be found on (Alon et al., 2019)).

4 EVALUATION

In this section, we will discuss the evaluation process to detect XSS using deep learning (Section 4.1) and the evaluation results (Section 4.2) obtained by using validation sets and test sets to select the best models.

4.1 Evaluation Process

To evaluate the XSS detection models using the generators presented in Section 2, we split each database into training, validation, and test sets. The number of safe and unsafe samples in each dataset is shown in Table 3. To avoid having misleading results because of possible similarities between the generated files, we take two actions. First, as stated in Section 2, we use a variable renaming scheme. Second, we split the databases using a mismatching distribution strategy. Specifically, we include in the training sets only files generated with rules 3 and 4; in contrast, files generated with the other rules were included in testing and validation sets (Table 3). Thus, we are evaluating not only the identification of XSS but the ability of the models to generalize the OWASP rules.

After splitting the databases, using the training-set and the validation-set, we train and evaluate both approaches obtaining the confusion matrix values (i.e., FP, FN, TP, TN) and computing the related metrics (e.g., accuracy, precision, recall, f-measure). This process is achieved for D1 and D2 databases for the Concatenation representation approach and D2 for the hashed-AST-based representation approach, as explained in Section 3.2. We repeat this process several times since we experiment with different hyper-parameters for both approaches.

In the Concatenation representation approach, we use two hyper-parameters: the embedding token size and the number of dense layers. For D1 and D2, we create six binary classifier models constituted by several dense layers with a different number of neurons, namely:

- 64-128-64-1
- 64-128-256-128-64-1
- 64-128-256-512-256-128-64-1
- 128-256-128-1
- 128-256-512-256-128-1
- 128-256-512-1028-512-256-128-1

We vary the token vector size for each of these six binary classifier models from 100, 200, 300, 500 to 800 spaces.

Regarding the hashed-AST-based representation approach, we evaluate it by varying three hyper-parameters: the maxContext, the maxPath width and the maxPath length. We vary the maxContext from 100, 200, 300, 500 to 800 spaces. Concerning the maxPath width, and maxPath length, we experiment with 10, 20, 30, 50, 80, and 130 spaces.

In this way, we train and test 150 models (30 models for each approach's database) and evaluate them by measuring the metrics previously mentioned.

After training and evaluating the approaches, we select the best evaluation configuration for each approach's database - as summarized in Table 4. To do

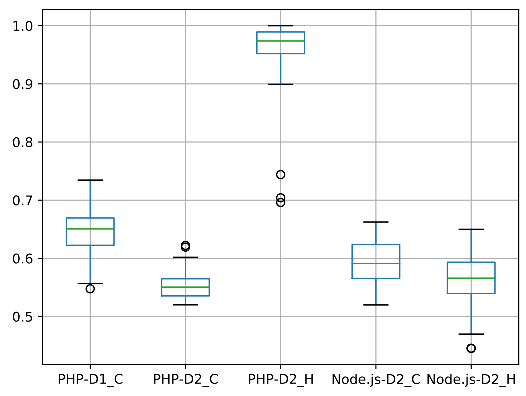


Figure 5: Validation-set Results.

this, we select the model with the highest recall from a subset of those models with precision over the average (85% for the Hashed-AST approach and 65% for the Concatenation approach). We added this refinement because, in an ideal model, the best vulnerability detection system will not miss any vulnerabilities - i.e., the recall metric would be equal to 1 - nor trigger false alarms - i.e., the precision metric would be equal to 1 -, which means f-measure equals 1.

Using these best configurations, we re-evaluate the five models using the test set. The metric results for the re-evaluation of these best configurations are presented in Table 6 and described in Section 4.2.2.

4.2 Evaluation Results

In this section, we present the results of the evaluation. First, we present the results[19] obtained using the validation sets (see Section 4.2.1). Using the validation sets, we tuned the hyper-parameters of the approaches to finally select the best model for each approach-database pair. The best models were then re-evaluated using the test sets whose results are presented in Section 4.2.2.

4.2.1 Validation-set Results

Figure 5 shows the recall results of the models during the validation phase in which the hyper-parameters were tuned. Regarding PHP, the results for the Hashed-AST approach using the database PHP D2 are visibly better than other cases with an average recall of 94.67% (std. dev. 8.2%). In Node.js, the Concatenation approach obtained an average recall of 59.32%

[19]Because of space constraints all the results can be accessed online at https://bit.ly/32wWbZS

Table 4: Summary of the best model configurations chosen after the evaluation.

Language	Validation set	Concatenation (NLP) hyperparameter		Hashed-AST (PLP) hyperparameter		
		embedding-size	layers	maxPath width	maxPath length	maxContext
PHP	D1-HTML	500	64-128-256-128-64-1	-	-	-
	D2-echo-HTML	800	128-256-512-1028-512-256-128-1	20	20	800
Node.js	D2-write-HTML	800	128-256-512-1028-512-256-128-1	80	80	800

(std. dev. 3.95%), while the Hashed-AST approach got 55.71% (std. dev. 5.71%).

However, to claim any statistically significant difference in the recall values obtained by the approach-database pairs, a statistical test is needed. Specifically, we employ the Kruskal-Wallis non-parametric test with a probability of error of $\alpha = 0.05$. After running the test, we obtained a p-value = 1.77E-21. It means that there is a significant difference between at least two pairs. To analyze which approach-database pairs are significantly different between them, we conducted Pairwise Wilcoxon Rank Sum Tests post-hoc comparisons. The post-hoc tests revealed that there is a significant statistical difference to claim that the Hashed-AST approach using the database PHP D2 is better than the other pairs. Also, we found a significant difference to claim that the Concatenation approach gets higher recall values using the database PHP D1 than PHP D2. However, we could not find any significant statistical difference between the Concatenation approach and the Hashed-AST approach when using the database Node.js-D2.

We also analyzed the variation of the recall value regarding the different configurations of the hyperparameters. In the case of Hashed-AST, we found a direct correlation between the recall and the maxContext parameter. Thus, the higher the maxContext value, the higher the recall. In the case of Node.js, we found that after reaching a value, the recalls start to decrease. However, we did not found a noticeable incidence of the maxPath width/length parameter, except by very low numbers. That is to say, with maxPath values of 10, the recall tends to be very low. In the Concatenation approach, we found that the parameter that most affects the recall is the embedding size. Thus, the higher the embedding-size set, the higher the recall. This relation is especially evident when using PHP D1. However, we found that the recall value tends to stabilize with an embedding size of 500 for the PHP D1 database, and 300 for Node.js. In PHP D2, we could not found any visible pattern regarding how the parameters affect the recall value. For example, with a neural network of 64-128-256-512-256-128-64-1 and 128-256-128-1, the recall value seems to reach the highest value with an embedding size of 100. In these cases, the recalls decrease as the embedding size increases. On the contrary, with a network of 128-256-512-1028-512-256-128-1, the recall increases as the embedding size increases.

As described in Section 4.1, we finished this phase by selecting the best model configuration for each approach-database-pair. These configurations were re-evaluated using the test set.

4.2.2 Test-set Results

The results of the best models selected in the previous step are summarized in Table 6. This Table is divided into two parts. The first one describes all the results that allow us to analyze the performance of the best models. The second part details the performance of the static analysis tools (Section 5) for the test set. For each approach-database pair, we report the confusion matrix and the associated metrics (i.e., precision, recall, etc.).

As shown in Table 6, the accuracy of the Concatenation representation approach is around 70%-73% for the different databases being the case of PHP D1, the highest accuracy for this approach (73.3%). PHP D1 also obtained the best recall for the Concatenation approach (73.5%). Regarding the precision, the highest values for this approach were obtained for PHP D2 (72.4%) and Node.js-D2 (72.8%).

Regarding the Hashed-AST approach, it presents the best results in almost all the metrics analyzed. The approach obtained an accuracy of 95.38% for PHP D2 and 79.7% for Node.js D2. The approach also obtained precision values close to 90% for both databases (92.4% and 89.4%). However, we noticed an important difference in the recall values. While a perfect recall was obtained for PHP D2 (99.9%), a recall of 63.2% was obtained for Node.js D2.

In summary, after analyzing precision, recall, and accuracy metrics, the best results are obtained for the Hashed-AST approach (PLP). However, in the absence of an appropriate parser to build models for heterogeneous code (that mixes two syntaxes), we could not evaluate the PHP D1 dataset for the Hashed-AST approach. Interestingly, the Concatenation approach (NLP) obtained its best results using the heterogeneous code of PHP D1.

5 COMPARISON

In this section, we compare our analyzers with XSS static analyzers for Node.js and PHP code. We select freely available tools that support a command-line interface, which facilitates the analysis's automation. Most of the tools that we test are referenced on the OWASP website[20].

5.1 Progpilot on PHP

We start by comparing our approaches with Progpilot, a static analyzer for PHP code vulnerabilities.

5.1.1 Progpilot Results for PHP D1 and D2

We run the Progpilot[12] tool on the D1 and D2 PHP databases to understand the capability of this tool to detect XSS on our dataset. As shown in Table 5, we measure the confusion matrix and other metrics such as accuracy, precision, TNR (True Negative Rate), recall, and F-measure to measure the tool's performance. The accuracy for D1 and D2 is 64.7% and 64.5%, with a precision of 76.2% and 76.1%, respectively. Progpilot has a good TNR that is over (92%). This means that Progpilot correctly returns a negative result for over 91% of samples without an XSS vulnerability. However, the recalls are relatively low (around 33%). To detect a security vulnerability, the recall is an important metric because it represents the number of correct XSS detections compared to the number of XSS that should have been found. Progpilot correctly returns a positive result for over 33% of samples. However, it incorrectly identifies as safe 67% of the unsafe samples. Thus, in our databases, Progpilot detects samples without XSS better than samples with XSS.

5.1.2 Progpilot Results for PHP Validation Set

To evaluate our models after the training and evaluation, we use the valid set data of D1 and D2. To compare with the metrics values that we obtain for our best deep learning model, we run Progpilot on the same valid set data of D1 and D2. The Progpilot's accuracy for the valid set D1 and D2 is 69.5% and 68.8%, with a precision of 78% (Table 6). Progpilot has a good TNR that is around 92%. This means that Progpilot correctly returns a negative result for over 92% of the samples that do not have an XSS. However, the recalls are relatively low (39.4%) and 37%. Thus, less than 40% of XSS unsafe samples are detected by the tool.

[20]https://owasp.org

5.1.3 Comparison between Progpilot and DL

Comparing the accuracy, precision, recall, and F-measure results of each of our models, they are significantly better than the Progpilot's results on PHP valid set data for the three databases (Table 6).

We can conclude that the deep learning approach gives better results than Progpilot.

5.2 AppScan Results for Node.js

We tried many free and open-source static analysis tools on Node.js databases such as nodejscan[21], insider[22], mosca[23], drek[24], devskim[25], and graudit[26]. However, none of them can detect XSS vulnerabilities in the train, test, and validation datasets in a significant way. Hence, these results are not included in Table 6.

We also tried a free scanning using a commercial tool called AppScan[27] on the Node.js validation dataset. As shown in Table 6, the AppScan's accuracy is 45.9%, with a precision of 45%, which is less than the deep learning model results. Also, the AppScan tool detects 3,724 of 3,725 safe samples as unsafe. Contrary to Progpilot, this tool does not distinguish between a secure sample and an unsecured sample on this validation database. On the other hand, the recall is higher (99%) because all the unsafe samples were detected as unsafe.

In summary, if we compare the accuracy, precision, recall, and F-measure results of each deep learning tested model, they are significantly better than the AppScan results on the Node.js validation database.

We can conclude that our Node.js XSS analyzers based on deep learning give better results than AppScan and six free open-source static analysis tools for the Node.js validation database.

6 LIMITATIONS

The evaluation conducted in this paper has several limitations that need to be taken into consideration. First, while our databases have representative examples of all the rules defined by OWASP, it only considers synthetic data. Building a dataset of real examples and applying the approaches presented in this

[21]https://github.com/ajinabraham/njsscan
[22]https://github.com/insidersec/insider
[23]https://github.com/CoolerVoid/Mosca
[24]https://github.com/chrisallenlane/drek
[25]https://github.com/microsoft/DevSkim
[26]https://github.com/wireghoul/graudit
[27]https://www.hcltechsw.com/products/appscan

Table 5: Progpilot results for each PHP database.

Database	Total_safe	Total_unsafe	TP	FP	TN	FN	Accuracy	Precision	TNR	Recall	F-measure
D1 HTML	23,200	12,544	3,584	1,120	11,424	7,072	0.647	0.762	0.911	0.336	0.466
D2 echo-HTML	23,200	12,544	3,518	1,102	11,442	7,138	0.645	0.761	0.912	0.330	0.460

Table 6: Comparison of deep learning best models results with static analyzer for the test dataset.

Methods	Language	Testing set		Accuracy	Precision	Recall	F-measure	TNR	TP	FP	TN	FN
Concatenation (NLP)	PHP	D1	HTML	0.733	0.68	0.735	0.70	0.732	2,079	974	2,662	749
		D2	echo HTML	0.707	0.724	0.534	0.615	0.842	1,510	576	3,060	1,318
	NodeJs	D2	write HTML	0.720	0.728	0.631	0.676	0.800	2,001	748	2,929	1,170
Hashed-AST (PLP)	PHP	D2	echo HTML	0.9538	0.924	0.999	0.918	0.868	2,760	494	3,241	1
	NodeJs	D2	write HTML	0.797	0.894	0.632	0.740	0.937	1,999	236	3,489	1,164
Progpilot (static)	PHP	D1	HTML	0.695	0.780	0.394	0.524	0.918	1,088	307	3,428	1,673
		D2	echo HTML	0.688	0.780	0.370	0.502	0.923	1,022	289	3,446	1,739
AppsScan (static)	NodeJs	D2	write HTML	0.459	0.450	0.990	0.610	0	3,163	3,724	1	0

paper could present new challenges that should be addressed in future work. For example, since the recollection of a large real dataset could be impractical, some deep learning techniques to reduce the size of the dataset could be used. One possibility could be the use of transfer learning. Another possibility to mitigate this threat, that we employed in this paper, is to train with mismatching distribution. However, using real data, it would be possible to train with a large synthetic dataset and evaluate it with a smaller, real one.

Another limitation is that currently, we are analyzing single files. We acknowledge that this is not representative of most real cases. For this reason, the representation approaches should be extended in future work to take into account the invocations between different files. Also, a limitation of the current representations is that the size of the representations is directly correlated with the length of the source code file. This could harm the applicability of the approaches in large code fragments. Designing size-efficient representations is a complex task and beyond the scope of this work.

7 RELATED WORK

We have already discussed in the introduction the positioning of XSS static detection methods concerning dynamic methods and black box scanners/fuzzers and discussed the positioning of our results with popular static analysis tools for PHP and Node.js in Section 5. Different approaches for code representations were discussed in Section 3. This section focuses on the closest works related to deep learning for the detection of security vulnerabilities.

VulDeePecker (Li et al., 2018) is a prototype to detect buffer errors (CWE-119) and resource management errors (CWE-399) related to library/API function calls on C and C++ source code using BLSTM neural networks. VulDeePecker uses three steps to design a pre-processing step to create a vectorial representation of C/C++ source code. First, the approach extracts library/API function calls with the Checkmarx commercial tool and extracts a list of program slice instructions semantically related. After that, the approach labels each of them and uses symbolic representation to tokenize each part the program slices. The tokens are inputted to the Word2Vec algorithm to generate vectors of the symbolic representation of each program slices. VulDeePecker used two datasets maintained by the NIST and the SARD project related to buffer error (i.e., CWE-119) and resource management error (i.e., CWE-399) in C/C++ source code. Our work shares with VulDeePecker the goal of static analyzing and detecting security vulnerabilities by using deep learning. Whereas VuldeePecker analyzes CWE-119 and CWE-399 for and C and C++, we analyze XSS (CWE-79) for PHP and Node.js. The two code representations that we evaluate in our work are different from the code gadget representations used by VudeePecker. Differently to us, VulDeePecker extracts library/API function calls as key points to generate a set of code semantically related to each other by using Checkmarx to create code gadget.

Neutaint (She et al., 2020) is a prototype to dynamically track information flow from sources to sinks using neural program embeddings and fully connected layers on the model architecture. It uses AFL fuzzer on several programs to generate a list set of couples associated sources and sinks. Neutaint creates a vectorial representation of specific programs. Instead of representing statically source code in vectors, this prototype predicts the corresponding taint sinks for the specified program to set of taint sources. Compare to classical dynamic taint analyses, the information flows tracked by Neutaint are not the information flow given by the execution of the program. However, the information flows inside the neural network with the generation of saliency maps. Our work shares with Neutaint the goal of finding incorrect flow from sources to sinks. However, Neutaint is essentially a dynamic tool, based on (the learn behavior of) execution of programs. In contrast, our work does not execute programs but learns from the code syntax, as in classical static program analyses.

Other tools using dynamic techniques include the Blender prototype (Wang and Su, 2020) that used the COSET dataset consisting of programs developed by programmers while solving ten coding problems and collected the execution trace by running each program with several randomly generated inputs. Blender used several attention layers with encoder-decoder neural architecture. An encoder neural network reads and encodes a source sentence into a vector-based on which a decoder outputs a translation. Like Code2vec, this prototype tried to predict method name by using an attention layer mechanism. Mitch (Calzavara et al., 2019) uses a browser extension HTTP-Tracker to manually label HTTP requests sent from web applications as sensitive or insensitive HTTP requests to detect CSRF attacks. The HTTP requests database was created by selecting Alexa ranking websites featuring authenticated access. Mitch uses different kinds of binary classifiers such Logistic Regression (LogReg), Support Vector Machines (SVM), Decision Trees (DT), Random Forests (RF), and Gradient Boosted Decision Trees (GBDT). In Mitch's approach, features are manually selected. Thus, this task required a significant amount of human effort, backed up by strong domain knowledge, to design 49 features representing three properties of the HTTP request, which are structural, textual, and functional. In contrast to our work, we used deep learning techniques to obtain features automatically.

8 CONCLUSION

In this work, we explore static approaches to detect XSS (stored and reflected) vulnerabilities using neural networks. We built two different code representations based on NLP and PLP and generate models using different neural network architectures for static analysis detection in PHP and Node.js using a mismatching distribution strategy. We evaluated our models in two datasets of PHP code: one including HTML/JavaScript as code and one including it as text. The Hashed-AST representation provides good performance for PHP and Node.js code. The Concatenation representation gives us good results for both programming languages but has a better recall when the HTML/Javascript is included as code, such as the case of PHP.

Using the generated PHP and Node.js datasets, we compared our results with a well-known static analyzer for PHP code, ProgPilot, and a known scanner for Node.js, AppScan static mode. Our analyzers using neural networks overcome the results of existing tools in all cases.

Notice that our generators can be used independently of our study and our models can also be applied to different security vulnerabilities (e.g. CSRF, DOM-XSS, etc) if the corresponding datasets are available.

As future work, we plan to extend this work to the analyses of code distributed in several files in a web application (using interprocedural analyses) as well as to different vulnerabilities that can be included as part of our generators.

Acknowledgment: This research has been partially supported by the ANR17-CE25-0014-01 CISC project, STICAmSud project 18STIC- 02, the Inria Project Lab SPAI, and the European Unions Horizon 2020 research and innovation programme under grant agreement No 830892. We thank anonymous Secrypt reviewers for their work, David Balazarotti and Benjamin Grégoire for suggestions related to preliminary results of this work. We thank Thorsten Holz for his help in finding datasets, to Manuel Serrano for his suggestions on AST representations, and to the SAMATE project at NIST for providing the initial PHP generator that is the base of this study.

REFERENCES

Allamanis, M., Brockschmidt, M., and Khademi, M. (2018). Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Allamanis, M., Peng, H., and Sutton, C. (2016). A convolutional attention network for extreme summarization of source code. In Balcan, M. and Weinberger, K. Q., editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*. JMLR.org.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL).

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society.

Calzavara, S., Conti, M., Focardi, R., Rabitti, A., and Tolomei, G. (2019). Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE.

Chlipala, A. (2016). Ur/web: a simple model for programming the web. *Commun. ACM*, 59(8).

Doupé, A., Cova, M., and Vigna, G. (2010). Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Kreibich, C. and Jahnke, M., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings*, volume 6201 of *Lecture Notes in Computer Science*. Springer.

Fogie, S. (2007). *XSS Exploits: Cross Site Scripting Exploits and Defense*.

Gundy, M. V. and Chen, H. (2009). Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society.

Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., and Veanes, M. (2011). Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association.

Lekies, S., Kotowicz, K., Groß, S., Nava, E. A. V., and Johns, M. (2017). Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In Thuraisingham, B. M., Evans, D., Malkin, T., and Xu, D., editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS.

Livshits, B. and Chong, S. (2013). Towards fully automatic placement of security sanitizers and declassifiers. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM.

Luo, Z., Rezk, T., and Serrano, M. (2011). Automated code injection prevention for web applications. In Mödersheim, S. and Palamidessi, C., editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*. Springer.

Melicher, W., Das, A., Sharif, M., Bauer, L., and Jia, L. (2018). Riding out domsday: Towards detecting and preventing DOM cross-site scripting. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In Bengio, Y. and LeCun, Y., editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Samuel, M., Saxena, P., and Song, D. (2011). Context-sensitive auto-sanitization in web templating languages using type qualifiers. In Chen, Y., Danezis, G., and Shmatikov, V., editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. ACM.

Saxena, P., Molnar, D., and Livshits, B. (2011). SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In Chen, Y., Danezis, G., and Shmatikov, V., editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. ACM.

Schoepe, D., Balliu, M., Pierce, B. C., and Sabelfeld, A. (2016). Explicit secrecy: A policy for taint tracking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE.

She, D., Chen, Y., Shah, A., Ray, B., and Jana, S. (2020). Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE.

Staicu, C.-A., Pradel, M., and Livshits, B. (2018). SYNODE: Understanding and automatically preventing injection attacks on NODE.JS. In *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS.

Wang, K. and Su, Z. (2020). Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.