Run-time Detection of Cross-site Scripting: A Machine-Learning Approach Using Syntactic-Tagging N-Gram Features

Nurul Atiqah Abu Talib *atiqah@hanyang.ac.kr*
Computer Science and Engineering
Hanyang University ERICA
Ansan, 15588, Republic of Korea

Kyung-Goo Doh *doh@hanyang.ac.kr*
Computer Science and Engineering
Hanyang University ERICA
Ansan, 15588, Republic of Korea

Abstract

Ensuring the security of web applications against cross-site scripting is practically a never-ending story. With the emergence of new applications with loaded payloads of open expressiveness and versatile functionalities to provide users with interactive services, the fight is even more challenging. A new feasible approach now in growing prominence is to use machine-learning classification. In this paper, we demonstrate an approach for payload abstraction through the translation of payloads into sentences of syntactic tags. This is to extract a normalized set of features of appropriate data and to minimize the problems of manually creating rules based on dangerous characteristics of payloads. We show that through abstraction and normalized features, we can accurately classify input payloads according to their proper categories. We assert that the security work is adequately informative to represent payloads and it can be more sustainable by using the automaton of machine-learning technique.

Keywords: Cross-site Scripting, N-gram, Web Application Security, Supervised Machine-learning, Tagging, Syntactic Structure.

1. INTRODUCTION

Despite the various security measures, the vulnerability of web applications to cross-site scripting (XSS) attacks remains an issue(MITRE Corporation, n.d.; OWASP, 2021). The client-based attacks causing unintended execution in a victim's browser are due to vulnerability related to insufficient input-validation(Pereira et al., 2020). Failing to prevent malicious scripts from entering or exiting the application could cause infiltration.

The vulnerability becomes more concerned with the advancement of web technology. Some applications now allow HTML input from users to incorporate advanced features. This is to expand the usability of web facilities or to create an enhanced user experience. Multiple interactive features of web pages, such as blogs or wikis, are a typical example of a case in point. Unfortunately, these features are taken advantage of by attackers to create new threats (Cure53, n.d.). Incidentally, they are in fact amassing formidability to the validation process to the point that legitimate strings are often mistaken as malicious(Gupta & Gupta, 2016). In any case, to do away with HTML inputs is to limit the application's functionality.

Various techniques proposed for the detection and prevention of XSS over the years include static analysis (Steinhauser & Gauthier, 2016; Yan & Qiao, 2016), dynamic analysis(Mitropoulos et al., 2016; Stock et al., 2014), and input validation(Bates et al., 2010; Gupta & Gupta, 2016; Rao et al., 2016). Static analysis techniques validate server-side code without executing it and can indicate flaws before the application is deployed. However, apart from requiring access to the

source code (Duraibi et al., 2019), this technique requires advanced security knowledge to validate the analysis report that has been known to produce false positives (Talib & Doh, 2021a). Dynamic analysis techniques validate source code during program execution and can minimize false positives reported during static analyses. However, it may incur runtime overhead as the number of inputs or the program size increases (Lou & Song, 2020; Mujawib Alashjaee & Duraibi, 2019). Input validation, such as XSS filtering techniques, inspects the input or output string to or from the application based on either a white list of allowed strings or a blacklist of prohibited strings. However, they are now seemingly less able to stand in control as security devices due to the ever-growing complexity of the influx of web application contents.

In short, these limitations are all the more reasons to call for the use of a new strategy in the protection of XSS vulnerabilities by detecting XSS payloads.

1.1 Motivation

As a way forward, we are looking at some new serious efforts to detect XSS attacks, specifically the use of the machine-learning technique. It is a technique now drawing particularly greater attention in the field of software security for its favorable property to automatically categorize new data from the existing data. It provides an added advantage of reducing manual dependency in the procedure.

In the application of machine learning, a classification model requires numerical inputs known as *feature vectors*. By a crucial process called *feature engineering*, the data features are extracted to inherently describe the dataset to represent the problem to the model. The model will then use these feature vectors to learn and solve the related problem.

In previous work, features are created based on malicious characteristics of payloads (or web pages) and are extracted using customized string-matching algorithms(Fang et al., 2018; Nunan et al., 2012; Rathore et al., 2017). However, this approach is somewhat constricting. First of all, the manual feature-engineering process that requires human intervention and specialized knowledge of XSS may not expand well with the new advancement in web technologies. Secondly, the focus on malicious characteristics as features may potentially preclude benign characteristics that may be substantial information for classification. Thirdly, the use of string-matching algorithms may misidentify the correct meaning of a string in the payload.

These limitations provide us with a motivation to attempt a way of solving by translating payloads into a form of text-based features based on their syntactic structure for classification. Because text classification is proven to be effective in natural language processing (NLP), such as spam filtering, intrusion detection(Nasser Mohammed & Mohamed Ahmed, 2019), and sentiment analysis, its use can be extended to the task of identifying malicious or benign input payloads in web applications.

Overall, this technique may help to reduce the arbitrariness that comes with the concrete language of payloads, and therefore, improve a model's detection capability. It may also help to potentially minimize the effort of provisionally selecting features by allowing the model to automatically learn the words pertaining to a class. In other words, without the need to manually define a blacklist of features specific to malicious payloads, we believe that our classification model is able to automatically learn a solution based on the information provided regardless of a malicious or benign class.

1.2 Contributions

We present this case here to offer an alternative XSS detection technique by classifying payloads using text-based features. It is the features for the classification process to replace the blacklist-only approach of extracting features by means of string-matching algorithms. Using a supervised machine-learning method, we attempt to develop an approach of creating an improved automatic classification of XSS in web applications by using syntactic structures of payloads in the feature set. By this approach, we offer the following contributions:

- A pre-processing method of translating payloads into syntactic-structure-aware documents using an HTML and a URI parser.
- A feature extraction approach using n-grams of a normalized set of features.
- A novel XSS-attack detection methodology that has achieved satisfactory results in terms of accuracy, precision, recall, and F1 rate with various classifiers.

Following are the discussions of our work in 5 sections. Section 1 contains an introduction of the study as is already discussed. In Section 2, we present some background concepts, while Section 3 outlines our methodology. The results of our work are explained in more detail in Section 4. Section 5 contains the conclusions.

2. BACKGROUND

This section overviews XSS and its vulnerabilities, reviews the literature on feature engineering to detect XSS, and summarizes the use of n-gram in text classification for NLP.

2.1 Cross-site Scripting

XSS is among the top security threats in web applications for more than a decade(OWASP, 2004, 2021). It occurs when attackers inject malicious scripts into vulnerable applications to be executed in a victim's browser. To illustrate, consider the following benign URI with the user input payload "John" (underlined) and its syntactic structure:

`http://www.safe.com/welcome_page.html?username=John`
Syntactic type: Text

An attacker can successfully insert a malicious script as input to the application if there is no proper validation as follows:

`http://www.safe.com/welcome_page.html?username=<script>doEvil()</script>`
Syntactic type: HTML script element

Notably, it is common that malicious and benign payloads may have different syntactic structures (Lekies et al., 2013).

XSS is an attack often associated with HTML injection due to the common presence of both languages (scripting and HTML) in the attack payload (Rao et al., 2016). The inclusion of malicious scripts in HTML as payloads can be done in various ways (van Oorschot, 2020). We categorize them as HTML elements and attributes as follows:

1) HTML Elements

- Inline script tags,
  e.g., `<script>doEvil()</script>`
- External scripts,
  e.g., `<script src="http://malicious-site.com/xss.js"></script>`
- CSS or style tags,
  e.g., `<style>@import url("http://malicious-site.com/xss.css");</style>`

2) HTML Attributes

- CSS or style attributes,
  e.g., `<div style="background-image:url(javascript:doEvil())">`

- URI attributes,
  e.g., `<img src="javascript:doEvil()" >`and
  `<a href='http://safe.com?name=%3Cscript%3EdoEvil()%2Fscript%3E'>link</a>`
- Event handlers,
  e.g., `<img src='img.jpg' onload='doEvil()' />`

In point of fact, malicious payloads are often crafted by attackers to exploit XSS vulnerability contexts of a target application. By contexts we mean the embedding location of the payload in the application (Kallin & Lobo Valbuena, n.d.; Lekies et al., 2013), such as the following:

- HTML element-content value,
  e.g.,`<p>untrusted_input</p>`
- HTML attribute value,
  e.g., `<p id='untrusted_input'>`
- URI query value,
  e.g., `<a href='http://site.com?query=untrusted_input'>`
- CSS value,
  e.g., `<p style="color:'untrusted_input';">`and
  `<style>untrusted_input</style>`
- JavaScript value,
  e.g., `var a = 'untrusted_input';`and
  `<script>untrusted_input</script>`

The `untrusted_input` represents the place where a payload is to be embedded. Note that the payload can be embedded within a context with or without quotes (single or double). In some cases, the embedding contexts are not exploited by injecting a complete element. Rather, only a partial string is injected to abuse the tags or attributes that are already present in the surrounding context. This relates to the act of element hijacking, unquoted attribute injection, trailing content, etc. (Gundy et al., 2009; Stock et al., 2014)

By all accounts, the safety measure is to scrutinize payloads that come as inputs into the applications. Then, it is to determine whether the inputs are benign, or otherwise, in the process of validation. In short, it is crucial that elements in the payload, be they scripts, style, other HTML tags, or attributes; all are subject to scrutiny. It follows that the legitimacy of the values associated with the elements should also be properly examined.

2.2 Feature Engineering for Detecting Cross-site Scripting

The feature engineering process for the detection of XSS primarily begins with examining malicious (XSS) or benign payloads. The ad-hoc extraction of features in previous works is to search for *dangerous characteristics* of known XSS payloads using customized functions(Fang et al., 2018; Habibi & Surantha, 2020; Mereani & Howe, 2018; Nunan et al., 2012; Rathore et al., 2017). The features, for example, are the payload length, the number of URL domains present, and the presence of special and/or encoded characters. These features would help differentiate between the following two payloads of different query values:

Payload	Class
`http://www.safe.com/index.html?lang=English`	Benign
`http://www.safe.com/index.html?lang=%3Cscript%20src%3Dhttp%3A%2F%2Fmalicious-site.com%2Fxss.js%3EdoEvil%28%29%3C%2Fscript%3E`	Malicious

The selection of features may also include syntactical or text-based features of dangerous strings, which are extracted using string-matching algorithms. For example, the presence of the "script" word (Habibi & Surantha, 2020) and the number of occurrences (Mereani & Howe, 2018;

Nunan et al., 2012; Rathore et al., 2017) or the embedding (Fang et al., 2018) of the script tags. This way of feature selection is often used in the domain of text classification, forming a vocabulary in the creation of a feature set based on the words in the corpus or dataset.

Extracting features following an ad-hoc manner and only searching for dangerous characteristics similar to a *blacklist-based approach* has elicited some limitations. For one, these selected features tend to exclude benign characteristics and thus, may potentially preclude substantial information associated with benign payloads. For another, they may succumb to new attack payloads that emerge with upgraded versions of HTML such as the attack via the embed tag succeeding the introduction of HTML5. Furthermore, the use of a string-matching approach for feature extraction may lead to the capture of incorrect data with the same pattern (Bates et al., 2010; Manico & Hansen, 2015; Talib & Doh, 2021b). An instance is the capture of a "script" word that may appear as a tag name or in the safe context of an element-content value.

As an alternative to the manual creation of blacklist-based features, the feature that we propose is a set that includes structural information of payloads that can be generalized to various payloads. Regardless of any upgrade to the HTML language, it covers both benign and malicious payloads.

2.3 N-gram in Text Classification

N-gram techniques are used predominantly for extracting features in the domain of NLP and information retrieval (Kowsari et al., 2019). The technique is a way to maintain the syntactical relation between words in the text representation. In traditional n-gram, the features to represent texts in a document is a collection of *n* adjacent words in the order of as they appear in texts.

The bag-of-words (BOW) technique is an instance of a 1-gram text representation. To illustrate, for the text, "*This is an example of a sentence*", the *1*-gram features that would represent it are: "*This*", "*is*", "*an*", "*example*", "*of*", "*a*", and "*sentence*". To add, the BOW technique disregards the grammatical, or syntactic, order of words in texts as it only describes their occurrences in a document. A common solution to this problem is to use *n>1*, such as *2*-gram. This way, the extracted features contain more information on the text's syntactic structure. The 2-grams (bi-grams) for the previous example are: "*This is*", "*is an*", "*an example*", "*example of*", "*of a*", and "*a sentence*".

3. METHODOLOGY

The methodology of our study subsumes the ensuing subsections: our research design of feature language and its application in the payload translation.

3.1 Feature Language

The focus of previous work for feature engineering has been answering the question of *selecting the strings that make payloads malicious*. In our attempt, we seek to address the question of *how payloads can be represented in a way that adequately maintains their syntactic information*. To answer, we examine the characteristics of input payloads that usually are HTML, URI, or textual strings. They are often embedded in the output web page to applications. As parts of a web page, they can be structured according to the HTML language and, therefore, be used to construct our language of features.

In our approach, the features are a pre-defined set of alphabetic tags each of which represents the syntactic function (*term*) of each sub-string (*chunk*) in the payloads. We refer our feature tags to *labels* as to avoid confusing them with HTML tags. Before we present our approach, we take into account the hierarchical strategies of normalizing payloads to obtain a sequence of labels.

The first attempt is the normalization of payloads whose labels are concrete values of chunks that results, by default, in *unnormalized* labeling. An example of unnormalized labelling is shown below:

Malicious #3	<div	id="div1"	onload="doEvil()">	Some text	
Unnormalized Labels	<div	id	div1	onload	doEvil()	Some text	

In opposition, the normalization is to use *high-level-labeling* of elements, attributes, and element-content values as features. However, it only provides very general information in classification. The examples of such normalization due to the same sequence of labels are shown in the following:

Benign #1	<a	id="anchor1"	href="http://safe.com?lang=en">	Link	
Malicious #1	<a	id="anchor1"	href="http://safe.com?lang= <script>doEvil()<script>">	Link	
Malicious #2	<script	id="script1"	type="text/javascript">	doEvil()	</script>
Malicious #3	<div	id="div1"	onload="doEvil()">	Some text	
High-level Labels	tag	attribute	attribute	content	endtag

To enhance normalization, the labeling is therefore to make only the concrete values of tags and attribute names distinct as labels. Examples of such labeling are div and a for `div`and `a`tags, respectively, and id and href for `id` and `href` attributes, respectively. However, by the presence of the various types of tag and attribute names in the HTML language, such labeling can lead to both data sparsity and expansiveness. The tag and attribute names include obfuscation that attackers use to evade detection and those new ones that come with the advancement of web technologies. Labeling them in such a way may also provide limited information because malicious and benign payloads can contain the same tag and attribute (Talib & Doh, 2021b). Examples of normalization from such labeling are shown in the following:

Benign #1	<a	id="anchor1"	href="http://safe.com?lang=en">	Link	
Malicious #1	<a	id="anchor1"	href=" http://safe.com?lang= <script>doEvil()<script>" >	Link	
Labels	<a	id	href	content	endtag

Malicious #2	<script	id="script1"	type="text/javascript">	doEvil()	</script>
Labels	<script	id	type	content	endtag

Malicious #3	<div	id="div1"	onload="doEvil()">	Some text	
Labels	<div	id	onload	content	endtag

Note that, although two of the malicious payloads have label sequences that differ from the benign, one of them remains equal.

To suit our purpose, we create feature labels based on two ruminations: (1) to assume payloads as context-embedded HTML strings, and (2) to consider the terms in the payload according to the HTML Document Object Model (e.g., element and attribute names, content values, etc.) and injection contexts (see Section 2.1). To be sure, we define features by merely taking into account generally important terms to represent payload structure and not solely basing them on malicious chunks. As such, we have labels of the types of terms categorized as follows:

- **HTML Tags**that refer to the`script`and`style`tags with their names as labels (script and style, respectively). This regards the JavaScript and CSS code injection contexts. All the other tags are commonly labeled tag.

- **HTML End Tags**that constitute the closing tags as a single label endtag to closely maintain the label of each chunk in the payloads.

- **Attribute Names** that represent the attribute names as a single label, attr, as they are regarded as mere carriers for exploit strings in malicious payloads (Talib & Doh, 2021b). This is also to avoid the problem causing data sparsity due to the variety of attributes, including event handlers.

- **Attribute Values** that comprise the attribute values as labels with respect to the URI components. The selected components as terms comprise: (1) the commonly used URI schemes, i.e., "http:" and "https:", (2) the script-related schemes, such as "javascript:" and "vbscript", (3) other schemes,(4) network locations, (5) file paths, (6) absolute paths, and (7) query names. The labels for these terms are urlscheme, scriptscheme, scheme, netloc, filepath, abspath, and query, respectively. To clarify, the attribute values can be generalized as either URI or non-URI strings (e.g., texts, JavaScript, or CSS code) according to the injection contexts. When creating normalized term labels by structuring strings in conformity to the URI grammar, non-URI strings are included in the URI components. They can be either other paths, query values, or fragments and are treated as other terms. Our consideration of all URI terms is because each of them can be referenced in vulnerable contexts.

- **Data** as those referring to the non-URI attribute values and non-HTML-element content values. They consist of alphanumeric text, integers, function calls (JavaScript code),and special character terms. Note, while we label the former 3 terms according to their stringed chunks, (text, int, and functioncall, respectively),we label special characters individually(e.g., `!` as excl,`"` as dquote, and `#` as hash). This is to give us more information on the underlying structure for any chunks with undefined terms. Examples of such cases are CSS or JavaScript code (other than function calls), or chunks in malformed, obfuscated, and encoded payloads. HTML element-content values are treated as other terms.

- **Other HTML Elements** as those representing directives (HTML doctype declarations or processing instructions) and comment elements with their respective labels, directive and comment. We label them as elements rather than according to their chunks due to their rare appearance as input payloads. In fact, there are only a few of these elements in our dataset, all of which are parts of the malicious payloads.

Lastly, our set of 50 selected features of labels is shown in Table 1. We would like to remind the readers that some of the labeling are one-to-one (e.g., `script` tags to script and the special character, `!`, to excl), while others are many-to-one (e.g., `img` and `a` tags to tag and `href`, `id`, `onload` attributes to attr).

Terms	Chunk Examples	Labels (Features)
Script Tags	`<script>`, `<script`	script
Style Tags	`<style>`, `<style`	style
Other Tags	`<a>`,`<a`,`<img>`, `<img`	tag
End Tags	`</script>`, `</style>`, `</a>`,`/>`	endtag
Attribute Names	`href`,`id`,`onload`	attr
URI Sche	`http:`,`https:`	urlscheme

mes		
Script-related Schemes	`javascript:,vbscript:`	scriptscheme
Other Schemes	`ftp:,mailto:`	scheme
Network Locations	`site.com:80, safe.com`	netloc
File Paths	`/path/index.html`	filepath
Absolute Paths	`/abspath`	abspath
Query Names	`page=,lang=`	query
Function Call	`doEvil(), foo(arg);`	functioncall
Special Characters	`!, ", #, $, %, &, ', (, ), *, +, ,,- ,. ,/, :, ;, <, =, >, ?, @,[, \, ], ^, _, `, {, |, }, ~`	excl,dquote,hash,dollar,percent,amp,squote,lpar,rpar,mul,add,comma,sub,dot,div,colon,semicolon,lt,equal,mt,ques,alias,lbrack,bslash,rbrack,caret,undersc,grave,lcurly,bar,rcurly,tilde
Integers	`0,999999`	int
Alphanumeric Texts	`var,i,id1, main,color`	text
Directive Elements	`<!DOCTYPE html>,<?...>`	directive
Comment Elements	`<!– … –->`	comment

TABLE 1: Features of 50 Selected Labels.

Having defined our features, the labels for the earlier examples are as follows:

Payload:	`<a`	`id=”anchor1”`	`href=”http://site.com?lang=en”>Link`						`</a>`
Labels:	tag	attr text	attr	urlscheme	netloc	query	text	text	endtag

Payload:	`<a`	`id="anchor1"`	`href="http://safe.com?lang=<script>doEvil()<script>">`	`Link`	`</a>`
Labels:	tag	attr text	attr urlscheme netloc query script functioncall script	text	endtag

Payload:	`<script`	`id="script1"`	`type="text/javascript">`	`doEvil()`	`</script>`
Labels:	script	attrtext	attr text div text	functioncall	endtag

Payload:	`<div`	`id="div1"`	`onload="doEvil()">`	`Some text`	`</div>`
Labels:	tag	attrtext	attr functioncall	text	endtag

We refer to the features in our approach as *normalized label* features. These features would be used to compute weights of labels that represent payloads by certain methods to form feature vectors (see Section 4.1). However, to closely maintain the order of labels in payloads, we use the bi-gram technique (refer to Section 2.3). It is of a lower bound to maintain the label order information.

3.2 Payload Translation

We translate payloads into a target language: *sentence* of labels. This is by applying syntax analysis through the use of a parser to scan the chunk patterns in payloads. It is an alternative to using lexical analysis techniques that may be insufficient for identifying terms, particularly in nested HTML elements.

For our purpose of translation, we utilize Python's HTML parser[1] by virtue of its essential capability to identify both HTML opening and closing tags. More so, it comes with *handler* functions to handle each HTML term in a payload. For instance, the *handle_comment* function handles comment elements. It also offers functions for processing named, decimal, and hexadecimal character references. Additionally, we add a URI parser[2] into the HTML parser to translate attribute values. Our customization of the parser is to implement the translation scheme.

The scheme starts with accepting a payload string and processes it using the HTML parser. The parser recognizes a chunk's term based on its pattern and passes the chunk to its respective handler function. The chunk representing an attribute value is passed to its handler and processed by the URI parser. Each handler function appends a label representing the chunk to the sentence string. For example, when the parser encounters a `script` tag, it appends the script label to the sentence. The chunks without a pre-defined term label (refer to Table 1) are passed to specific handlers for further processing to obtain their labels. To be sure, some attribute values of the URI components, specifically, "other paths", that are generally texts are passed to *handle_data*. Likewise, query values and fragments that can carry elements are passed to the start of the parser. In the event of an error while parsing, the parseerror label is appended to the sentence to mark its erroneous structure. The result of our scheme is a sentence.

[1] https://docs.python.org/3/library/html.parser.html
[2] https://docs.python.org/3/library/urllib.parse.html

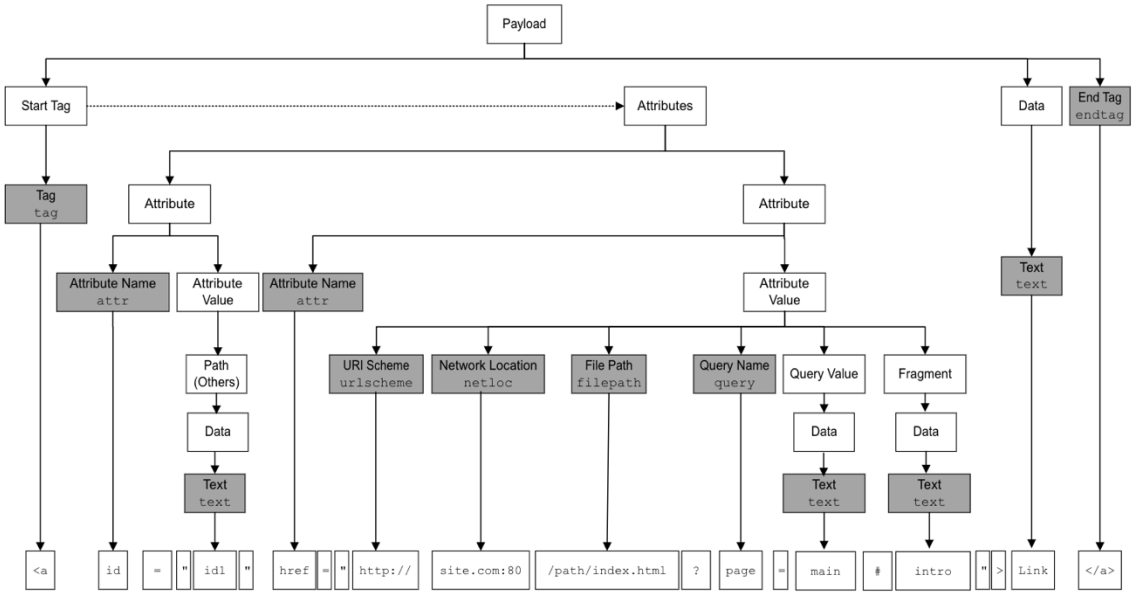

FIGURE 1: Syntactic Structure and Translation of a Hypothetical Benign Payload.

To provide a visual of how our translation scheme works, we present the syntactic structure of a hypothetical benign payload with its translation of labels (greyed boxes) as shown in Figure 1. The structure and translation of the same payload with a malicious script in the URI fragment are shown in Figure 2. To further illustrate our scheme with other examples, the syntactic structure and translations for different HTML elements (i.e., processing instruction, HTML doctype declaration, comment, and HTML tags) are shown in Figure 3. The same is shown in Figure 4, but with a `style` tag payload.

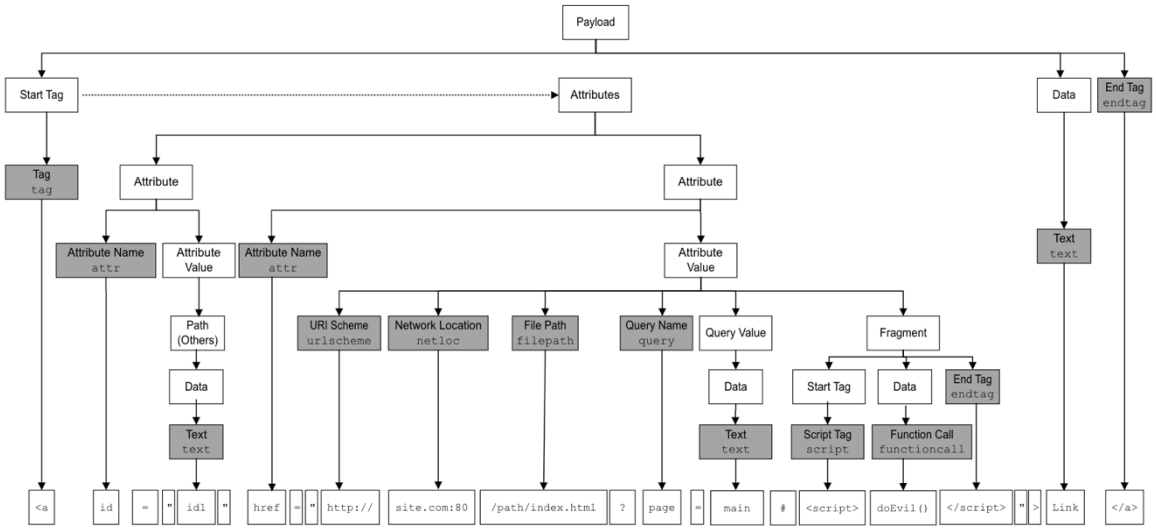

FIGURE 2: Syntactic Structure and Translation of a Hypothetical Malicious Payload.

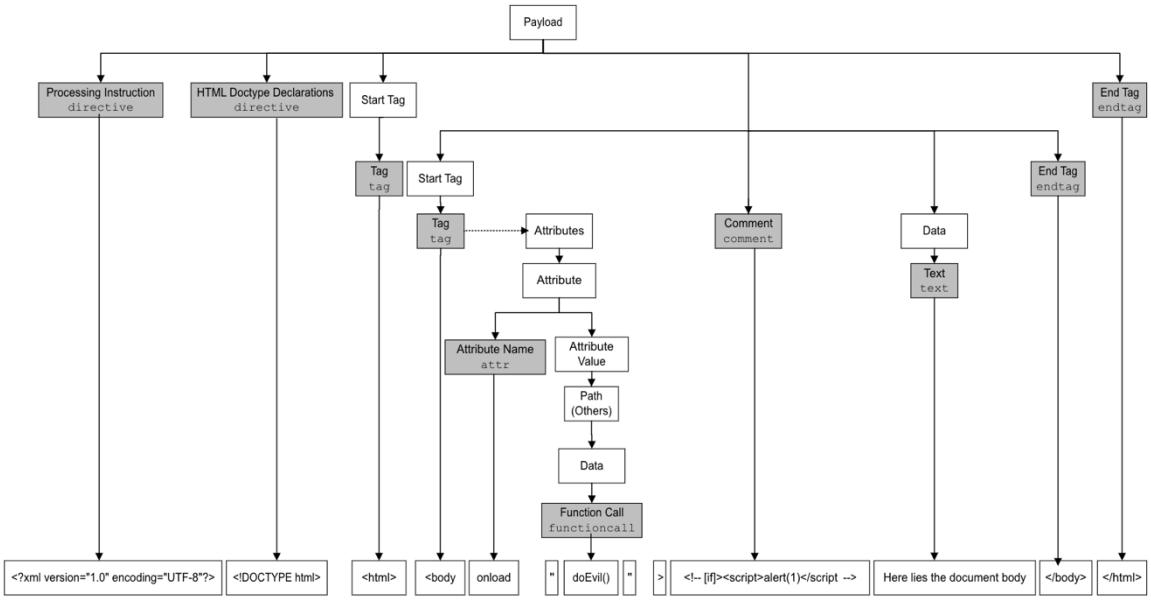

FIGURE 3: Syntactic Structure and Translation of Different HTML Elements.

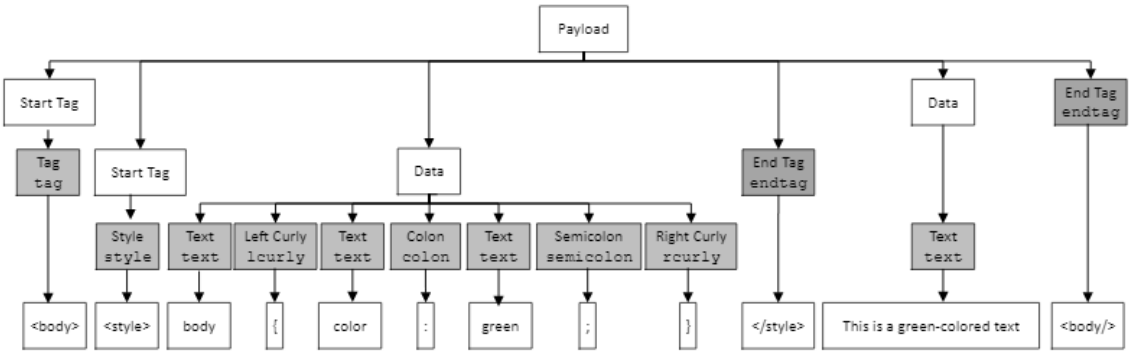

FIGURE 4: Syntactic Structure and Translation of a Hypothetical Style Tag Payload.

Note that in the case of sequential text, int, and special character labels in attribute or content values, we reduce them into a single label. An example of such a case is the chunk `This is a green-colored text`, where their labels of text text text text sub text text are reduced to only text sub text. Due to their high-frequent appearance in the values, it is to avoid *over-labeling*, i.e., unnecessary increase in weights of the label, that may negatively affect classification.

4. EXPERIMENTAL ASSESSMENT

In this section, we analyze and assess our methodology through an experiment using the real-world dataset previously used by other work using a classifier training and testing process. The details are as follows.

4.1 Preparation

The preparation involves basically the selection of vectorizers and datasets for our experiment. The sentence of labels is first transformed into a machine-readable vector using a process called vectorization. In this experiment, we use the TF-IDF[3] vectorizer in Python's Scikit-learn[4] library.

[3] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

Its weighting algorithm can produce a vector based on word counts. To apply bi-gram features, we set TF-IDF's `ngram_range` parameter with the value of `(2,2)`. This specifies the lower and upper boundary of the range of `n`-values, respectively.

Our collection of the dataset is chosen from the work of Fang et al. (2018) which consists of 64,833 payloads. The 31,407 benign URIs originate from the DMOZ[5] database and the 33,426 malicious URIs originate from the XSSed[6] database. The target values in the dataset are the categorical data of 0 and 1 that are to represent the benign and the malicious outputs of a URI (payload), respectively.

4.1 Training and Testing

The experimentation is conducted using a Mac-Book Pro machine with 8GB of memory and a 2.3 GHz Dual-Core Intel Core i5 processor. As a procedure, the experimentation involves training and testing the classification model. In the training process, we start with translating a training dataset containing raw payloads as shown in Figure 5. This begins with each of the payloads individually fed to our translation scheme to result in sentences of features. The sentences are fed to the vectorization algorithm in order to develop a vector space (feature vectors) based on the bi-grams in the sentences. The vectors would then be used as an input to train the classification model. The final output is a trained classifier.

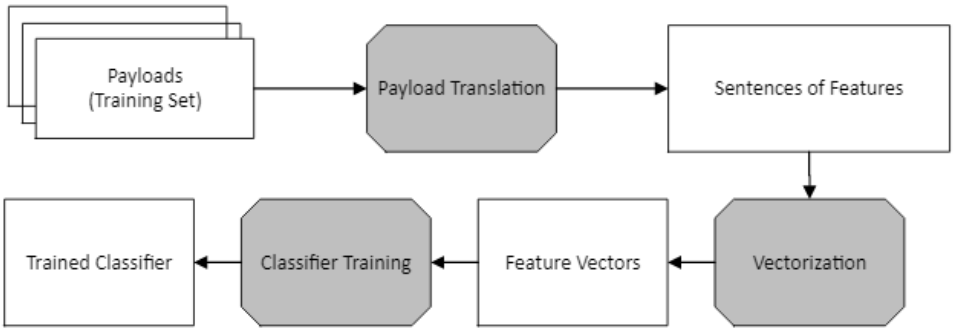

FIGURE 5: Overview of Training Process.

In the testing process, we utilize all the payloads in the testing set and prepare them for a translation and vectorization process. The classifier will next identify each payload's target value in the set to decide whether the payload's feature vector is either benign or malicious, based on the information it gained from the training phase.

We analyze the results of the study based on the accuracy, precision, recall, and F1 scoring metrics using the counts of true positives, true negatives, false positives, false negatives as follows:

- True positives (TP): The frequency of actual malicious payloads that are correctly classified as malicious
- True negatives (TN): The frequency of benign payloads that are correctly classified as benign
- False positives (FP): The frequency of benign payloads that are incorrectly classified as malicious
- False negatives (FN): The frequency of actual malicious payloads that are incorrectly classified as benign.

[4] https://scikit-learn.org/
[5] http://dmoztools.net/
[6] http://www.xssed.com/

- Accuracy: The ratio between correct detection and the total detection, *(TP+TN)/(TP+FP+FN+TN)*
- Precision: The ratio between the correct detection of malicious payloads and all the predicted malicious payloads, *TP/(TP+FP)*
- Recall: The ratio between the correct detection of malicious payloads and all the actual malicious payloads, *TP/(TP+FN)*
- F1: The harmonic mean between the precision and recall, *((2 × Recall × Precision))/((Recall+Precision))*

Additionally, we calculate the execution time (Time) for classification, which is measured as the amount of time spent by the classifier to make a decision on the testing set. Time is measured before the start and after the classification procedure, which includes the normalization and vectorization of each payload.

We use a stratified 10-fold-cross[7] validation in Python's Scikit-learn library to analyze the results of our approach. Nine folds are used for the training set, and one is used for the testing set. Each set contains the same distribution of malicious and benign payloads. This validation process is iterated 10 times.

In proceeding, we apply and compare 8 different trained classifiers to classify the dataset to validate their usability for the model in our approach. The classifiers are Random Forest, Neural Network, Decision Tree, Support Vector Machine (SVM), AdaBoost, K-Nearest Neighbors, Logistic Regression, and Naive Bayes. The selection of classifiers is conducted on their default parameters.

Classifier	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
Random Forest	99.25	99.82	98.72	99.26
Neural Network	99.19	99.86	98.56	99.21
Decision Tree	99.15	99.77	98.58	99.17
SVM	99.14	99.82	98.52	99.16
AdaBoost	99.04	99.58	98.55	99.06
K-Nearest Neighbors	98.97	99.62	98.38	99.00
Logistic Regression	98.70	99.17	98.29	98.73
Naive Bayes	95.67	99.72	91.86	95.63

TABLE 2: Classification results for different classifiers.

The classifier for evaluation of our approach is chosen based on the report of the classification results shown in Table 2. According to the results, with the exception of Naïve Bayes, all classifiers produce promising classification results, showing a similar percentage in accuracy, precision, recall, and F1. On this count, we can say that they are all usable in our approach. Hence, we simply choose the most promising classifier, Random Forest, for our evaluation.

4.2 Evaluation

To evaluate, we compare the results of our approach with (1) those of other text-based feature approaches from previous work and (2) those of other attempted features. The feature approaches are as follows:

a) **Script-related Concrete Features** are those that only take on the dangerous script-related concrete values following the discussion in section 2.2. We include HTML start and end tags, event handlers, function calls, and special characters in reference to a previous work (Fang et al., 2018). There is an average of 2,228 bi-grams in these features.

[7] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

b) **Script-related label features** are the single labels to each script-related concrete features (start tags, end tags, event handlers, function calls, and special characters). This feature set comprises an average of 26 bi-grams.

c) **Word-only features** are those consisting of only alphanumeric chunks. It is a result of applying the common pre-processing step of removing special characters in NLP to attain a sequence of words (Kowsari et al., 2019). Unfortunately, this set is excluded from our evaluation. This is because the machine used for the experiment was not capable of supporting the computational power required for the execution due to its large feature set. As it churns out a selection of 66,339 unigram features, it is an extremely too big a number of bi-gram features to go for a similar process as other approaches. This speaks for a reason to the problem in our evaluation of the unnormalized feature approach for XSS detection.

d) **High-level-labeling features** represent the normalization of elements, attributes, and element-content values as discussed in section 3.1. The set contains an average of 20 bi-gram features of the training set.

e) **Concrete Tags and Attributes (CTA) features** represent the normalization of elements that includes the tags and attribute names as labels as discussed in section 3.1. They are those consisting of the concrete values of HTML start and end tags, and attribute names. The set contains an average of 796 bi-gram features of the training set.

f) **Advanced Concrete Tags and Attributes (ACTA) features** represent the extension of CTA features to include attribute values. They are function calls, URI schemes, non-integer query values, and fragments. It is somewhat a concrete version of our approach with a selection of normalized labels (texts and some attribute values) to avoid the unnormalization problem in (c). The features contain an average of 9,753 bi-grams of the training set.

We then compare those results with our experimental results using **normalized label features**. Although our features include labels for dangerous and legitimate chunks, we manage to come up with a comfortable number of only 778 bi-grams comprising the 50 features presented in Table 1. Note that we include 2 additional labels to each feature set: (1) the document label that marks the start of the sentence to help form a bigram for payloads containing only a single chunk, and (2) the parseerror label (refer to Section 3.2).Following are the examples of the feature sentence used in the payload:

Payload	<img id = 'id1' src='x' onload = 'doEvil()' / >
Script-related concrete	<img onload doEvil() / >
Script-related label	tag event functioncall endtag
High-level-labelling	tag attr attr attr endtag
CTA	<img id src onload endtag
ACTA	<img id text src text onload doEvil() / >
Normalized label	tag attr text attr text attr functioncall endtag

Feature Approach	Performances				Time (sec.)
	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)	
Script-related concrete	51.56	51.56	100.00	68.03	1.72
Script-related label	96.62	98.58	94.83	96.67	1.22
High-level-labelling	92.45	100.00	85.35	92.09	0.50
CTA	92.44	100.00	85.34	92.09	0.61
ACTA	86.10	88.60	83.81	86.13	2.49
Normalized label	99.25	99.80	98.74	99.27	1.64

TABLE 3: Comparison of classification performance of feature selection approaches.

Table 3 shows the comparison results of classification models using 5 different features and the normalized features of our approach. We observe that the use of our normalized label features appears with the best result in accuracy and F1, by a difference that ranges from 3-48% and 3-31%, respectively. In terms of precision, our approach stands the third best with a score of 99%, just 0.2% behind high-level labeling and CTA features. By the same token, our approach scores a high of 98% in recall, second to script-related concrete with a difference of less than 2%.

We note that despite the high score in precision of high-level-labeling and CTA features, they report a lower score in accuracy, recall, and F1. This is to say that these models are good at classifying truly malicious payloads, but only for a small proportion of detection. This highlights that their use of normalization at only elements and attributes level is a case of under-fitting in their classification model.

As for the script-related concrete features, they perform with a perfect recall (100%) but slack with a significantly lower score in accuracy, precision, and F1. This means that although the model using dangerous-only strings can correctly classify most of the malicious payloads, they go at the expense of misclassifying benign payloads as malicious. Therefore, the model is said to be over-fitting.

Although the approach of using script-related label features can result in better performance, it cannot compete with ours. The fact that our labeling covers more terms than that of the script-related, we conclude that ours is better equipped to handle both benign and malicious payloads within a controlled set of features.

Meanwhile, the approach of using ACTA features shows considerably lower scores of accuracy, precision, and recall compared to our approach. This means that the use of concrete attribute values does not necessarily improve a model's performance, although it also provides a steady balance between precision and recall.

To point out, addition of concrete features for arbitrary chunks, as is the case for script-related concrete and ACTA, can negatively affect a model's classification capability (except recall in the case of script-related concrete features). In contrast, the use of normalized labeling of features, such as high-level labeling and script-related label features can increase a model's performance. In some cases, adding concrete features of a somewhat rational set of chunks, such as tags and attribute names in CTA, to the normalized labels can also increase a model's performance, particularly in precision. However, our normalized labeling with various terms is adequately informative to come up with a well-balanced performance. Thus, helping to differentiate between malicious and benign payloads.

It is interesting that the number of features of each approach does not significantly affect the models' performance. This is evident in the similar performance of the 26 script-related label features and our 778 normalized label features. Similarly, the 20 features of high-level-labeling has a comparable performance result with the 796 features of CTA. However, a large set of unnormalized features would require an immense amount of computational power for the execution that can be offset by using our use of normalized label features. This is because ours is appropriate insofar as the number just suffices the need for the classification process to distinguish the benign from the malicious payloads.

Table 3 also shows the execution time for classification. It is measured as the amount of time spent to translate, vectorize, and classify the testing set into the category of malicious or benign payloads. The results show that the advanced concrete feature approach takes the longest 2.5-second time, followed by the script-related concrete feature approach of 1.7 seconds on average to complete execution. Our approach of normalized label features comes next with 1.6 seconds, followed by script-related label, CTA, and high-level-labeling with 1.2, 0.6, and 0.5 seconds, each. However, the differences of time score of the approaches are practically negligible, showing 1 second or less.

4.3 Comparative Assessment with Other Work

For purpose of further verification, we include the result comparing the performance of our approach and a similar work. Fang et al., (2018)applied an XSS detection approach called DeepXSS via word2vec embedding and Long Short-Term Memory (LSTM) recurrent neural networks. A comparison of our approach with theirs is shown in Table 4. The result shows that our use of the normalized label feature comes out slightly better in all three scores. This would convince us that our approach can well be an alternative for a solution in detection technique. Apart, the high complexity of deep-learning models that require longer training and testing duration would be saved by using machine-learning models (Liu & Lang, 2019). Additionally,

Approach	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)
DeepXSS (Fang et al., 2018)	N/A	99.5	97.9	98.7
Normalized Label	99.3	99.8	98.7	99.3

TABLE 4: Comparison of performance with an alternative solution.

deep-learning models often require large datasets to produce good results (Xin et al., 2018). For instance, Ferrag et al., (2020) in their attempt to show how deep-learning models outperform machine-learning models for intrusion detection using2 different datasets consisting of about 15 million and 3 million each. Apruzzese et al., (2018) who work with the size of 500,000 dataset are not able to show that deep-learning models can outperform machine learning. In any case, obtaining large datasets with a variety of distinctive payloads in the field of cyber-security is not easy, particularly for XSS.

4.4 Discussion

Using concrete strings from payloads as features has resulted in an overall lower classification performance and could risk data sparsity. This stands in contrast to using dangerous concrete strings that often end up with model over-fitting. The use of labels for the concrete strings as features would help to improve a model's performance on the conditions that it is not over-generalizing the labeling as it can lead to model under-fitting. One other pertinent point is that using features with syntactic structural information of only HTML and URI labels is evidently a reliable model with high and well-balanced performance. This is to say, there is a practically equal detection rate of precision and recall. Most importantly, our normalized features have shown to be adequately informative to represent payloads, thus helping a model to differentiate the malicious from the benign.

It is particularly relevant to say that our approach of using syntactic structure, instead of white lists or blacklists of known strings, as features is helping to eliminate human manning of creating or updating features. This would especially be of help to those with or without the background knowledge of XSS.

As with the advancement of web technology and the continuous crafting of new malicious payloads by attackers to circumvent defense mechanisms, there is a growing need for a feature set that does not expand with the incoming of new types of payloads. We have developed a translation scheme that transforms payloads into language of syntactic features using an HTML and a URI parser. By using this language, our feature set acquires a controlled number of 778 features on average, at least about an18 to 66,000 count reductions from a partially normalized or unnormalized set. Our study offers an opportunity to many researchers, particularly those interested in using NLP and machine learning, for the protection of web applications against other attacks.

5. CONCLUSION

As a solution to the problem of having to manually create blacklist-based features in supervised machine learning XSS classification, we have proposed a pre-processing method of translating payloads into a text-based feature language comprising normalized syntactic tags.We have shown that our approach has done a satisfactory job of correctly classifying XSS payloads. We

have provided commendable proof that the use of our approach in text classification is able to successfully distinguish a variety of malicious payloads (64,833) from benign ones. More importantly, our attempt in letting the machine make classifications based on past observations of payloads' syntactic structure instead of provisionally crafted rules is an initiative to be considered as a timely switch.

The evaluation of our approach using Random Forest classifier on real-world datasets, including the results with 7 other classifiers has demonstrated that our approach is high in accuracy, precision, recall, and F1 rates. We are also able to show that our approach is effective in its detection task and thus, pointing to a more credible selection procedure in the process of preventing various XSS attacks. Additionally, we have included a systematic comparison with a previous study. Although the use of features in similar work against XSS is already very strong, we have empirically shown that our approach is indeed practical and that our proposed feature has vastly improved the classification performance.

By doing away with using a blacklist or white list, our approach is free from any need for manual updates. Its ability to handle HTML, URI, and textual input payloads, enables our proposed XSS detection to be applied to various parts of the web application architecture without the need of acquiring the application source code.It is particularly applicable as a proxy to the application server and the browser, or as a validation mechanism in input source or output functions to external processors (the database or browser).By these characteristics, our approach can be beneficial to web developers, with or without background knowledge in XSS to protect applications against XSS.

There are a few possible directions in which this research can be taken to further improve the current results. Firstly, the consideration of additional features according to the CSS and JavaScript language is a very likely step for future attempts. An alternative is the addition of features that have been used in other works, such as URL length and the number of domains. Secondly, testing our approach using different vectorizers with different n-grams is worth investigating. Lastly, considering the classification of output strings from the application is potentially another way to go.

6. REFERENCES

Apruzzese, G., Colajanni, M., Ferretti, L., Guido, A., & Marchetti, M. (2018). On the effectiveness of machine and deep learning for cyber security. *International Conference on Cyber Conflict, CYCON*, *2018-May*, 371–389. https://doi.org/10.23919/CYCON.2018.8405026

Bates, D., Barth, A., & Jackson, C. (2010). Regular expressions considered harmful in client-side XSS filters. *Proceedings of the 19th International Conference on World Wide Web - WWW '10*, 91. https://doi.org/10.1145/1772690.1772701

Cure53. (n.d.). *HTML5 Security Cheatsheet*. Retrieved March 7, 2017, from https://html5sec.org/

Duraibi, S., Alashjaee, A. M., & Song, J. (2019). A Survey of Symbolic Execution Tools. *International Journal of Computer Science and Security (IJCSS)*, *13*(6), 244–254.

Fang, Y., Li, Y., Liu, L., & Huang, C. (2018). DeepXSS: Cross site scripting detection based on deep learning. *ACM International Conference Proceeding Series*, 47–51. https://doi.org/10.1145/3194452.3194469

Ferrag, M. A., Maglaras, L., Moschoyiannis, S., & Janicke, H. (2020). Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. *Journal of Information Security and Applications*, *50*. https://doi.org/10.1016/j.jisa.2019.102419

Gundy, M. Van, Gundy, M. Van, Chen, H., & Chen, H. (2009). Noncespaces: using randomization

to enforce information ow tracking and thwart cross-site scripting attacks. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 1–18.

Gupta, S., & Gupta, B. B. (2016). XSS-immune: a Google chrome extension-based XSS defensive framework for contemporary platforms of web applications. *Security and Communication Networks*, *9*(17), 3966–3986. https://doi.org/10.1002/sec.1579

Habibi, G., & Surantha, N. (2020). XSS attack detection with machine learning and n-gram methods. *Proceedings of 2020 International Conference on Information Management and Technology, ICIMTech 2020*, 516–520. https://doi.org/10.1109/ICIMTech50083.2020.9210946

Kallin, J., & Lobo Valbuena, I. (n.d.). *Excess XSS: A comprehensive tutorial on cross-site scripting*. Retrieved March 22, 2017, from https://excess-xss.com/

Kowsari, K., Meimandi, K. J., Heidarysafa, M., Mendu, S., Barnes, L., & Brown, D. (2019). Text classification algorithms: A survey. *Information (Switzerland)*, *10*(4). https://doi.org/10.3390/info10040150

Lekies, S., Stock, B., & Johns, M. (2013). 25 Million Flows Later - Large-scale Detection of DOM-based XSS. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*, 1193–1204. https://doi.org/10.1145/2508859.2516703

Liu, H., & Lang, B. (2019). Machine learning and deep learning methods for intrusion detection systems: A survey. *Applied Sciences (Switzerland)*, *9*(20). https://doi.org/10.3390/app9204396

Lou, B., & Song, J. (2020). A Study on Using Code Coverage Information Extracted from Binary to Guide Fuzzing. *International Journal of Computer Science and Security (IJCSS)*, *14*, 200–209.

Manico, J., & Hansen, R. (2015). *XSS Filter Evasion Cheat Sheet*. https://owasp.org/www-community/xss-filter-evasion-cheatsheet

Mereani, F. A., & Howe, J. M. (2018). Detecting Cross-Site Scripting Attacks Using Machine Learning. *Advances in Intelligent Systems and Computing*, *723*, 200–210. https://doi.org/10.1007/978-3-319-74690-6_20

MITRE Corporation. (n.d.). *CVE Details: The Ultimate Security Vulnerability Datasource*. Retrieved March 20, 2019, from https://www.cvedetails.com/vulnerabilities-by-types.php

Mitropoulos, D., Stroggylos, K., & Spinellis, D. (2016). How to Train Your Browser: Preventing XSS Attacks Using Contextual Script Fingerprints. *ACM Transactions on Privacy and Security ACM Trans. Priv. Secur*, *19*(2), 1–31. https://doi.org/10.1145/2939374

Mujawib Alashjaee, A., & Duraibi, S. (2019). Dynamic Taint Analysis Tools: A Review. *Salahaldeen Duraibi & Jia Song International Journal of Computer Science and Security (IJCSS)*, *13*, 231.

Nasser Mohammed, M., & Mohamed Ahmed, M. (2019). Data Preparation and Reduction Technique in Intrusion Detection Systems: ANOVA-PCA. *International Journal of Computer Science and Security (IJCSS)*, *13*(5), 167–182. https://www.cscjournals.org/manuscript/Journals/IJCSS/Volume13/Issue5/IJCSS-1498.pdf

Nunan, A. E., Souto, E., Dos Santos, E. M., & Feitosa, E. (2012). Automatic classification of cross-site scripting in web pages using document-based and URL-based features. *Proceedings - IEEE Symposium on Computers and Communications*, 000702–000707. https://doi.org/10.1109/ISCC.2012.6249380

OWASP. (2004). *OWASP Top Ten 2004*. https://www.owasp.org/index.php/Top_10_2004

OWASP. (2021). *OWASP Top 10:2021*. OWASP. https://owasp.org/Top10/

Pereira, R. F., Silva, R. M., & Orvalho, J. P. (2020). Virtualization and Security Aspects: An Overview. *International Journal of Computer Science and Security (IJCSS)*.

Rao, K. S., Jain, N., Limaje, N., Gupta, A., Jain, M., & Menezes, B. (2016). Two for the price of one: A combined browser defense against XSS and clickjacking. *2016 International Conference on Computing, Networking and Communications, ICNC 2016*. https://doi.org/10.1109/ICCNC.2016.7440629

Rathore, S., Sharma, P. K., & Park, J. H. (2017). XSSClassifier: An efficient XSS attack detection approach based on machine learning classifier on SNSs. *Journal of Information Processing Systems*, *13*(4), 1014–1028. https://doi.org/10.3745/JIPS.03.0079

Steinhauser, A., & Gauthier, F. (2016). JSPChecker: Static detection of context-sensitive cross-site scripting flaws in legacy web applications. *PLAS 2016 - Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, Co-Located with CCS 2016*, 57–68. https://doi.org/10.1145/2993600.2993606

Stock, B., Lekies, S., Mueller, T., Spiegel, P., & Johns, M. (2014). Precise client-side protection against DOM-based Cross-Site Scripting. *Proceedings of the 23rd USENIX Security Symposium (SEC'14)*, 655–670. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/stock

Talib, N. A. A., & Doh, K.-G. (2021a). Static Analysis Tools Against Cross-site Scripting Vulnerabilities in Web Applications : An Analysis. *Journal of Software Assessment and Valuation*, *17*(2), 125–142. https://doi.org/10.29056/jsav.2021.12.14

Talib, N. A. A., & Doh, K. G. (2021b). Assessment of dynamic open-source cross-site scripting filters for web application. *KSII Transactions on Internet and Information Systems*, *15*(10), 3750–3770. https://doi.org/10.3837/tiis.2021.10.015

van Oorschot, P. C. (2020). Web and Browser Security. *Information Security and Cryptography*, 245–279. https://doi.org/10.1007/978-3-030-33649-3_9

Xin, Y., Kong, L., Liu, Z., Chen, Y., Li, Y., Zhu, H., Gao, M., Hou, H., & Wang, C. (2018). Machine Learning and Deep Learning Methods for Cybersecurity. *IEEE Access*, *6*, 35365–35381. https://doi.org/10.1109/ACCESS.2018.2836950

Yan, F., & Qiao, T. (2016). Study on the detection of cross-site scripting vulnerabilities based on reverse code audit. *Proceedings of the 17th International Conference on Intelligent Data Engineering and Automated Learning, IDEAL*, *9937 LNCS*, 154–163. https://doi.org/10.1007/978-3-319-46257-8_17