Journal of Web Engineering, Vol. 17, No. 1&2 (2018) 028–044

TEXT-MINING AND PATTERN-MATCHING BASED PREDICTION MODELS FOR DETECTING VULNERABLE FILES IN WEB APPLICATIONS

MUKESH KUMAR GUPTA [a]
Department of Computer Science and Engineering
Malaviya National Institute of Technology, Jaipur, Rajasthan, India
mukesh.iitb08@gmail.com

MAHESH CHANDRA GOVIL
Department of Computer Science and Engineering
Malaviya National Institute of Technology, Jaipur, Rajasthan, India
govilmc@gmail.com

GIRDHARI SINGH
Department of Computer Science and Engineering
Malaviya National Institute of Technology, Jaipur, Rajasthan, India
gsingh.cse@mnit.ac.in

Received June 17, 2017
Revised September 10, 2017

The proliferation of technology has empowered the web applications. At the same time, the presences of Cross-Site Scripting (XSS) vulnerabilities in web applications have become a major concern for all. Despite the many current detection and prevention approaches, attackers are exploiting XSS vulnerabilities continuously and causing significant harm to the web users. In this paper, we formulate the detection of XSS vulnerabilities as a prediction model based classification problem. A novel approach based on text-mining and pattern-matching techniques is proposed to extract a set of features from source code files. The extracted features are used to build prediction models, which can discriminate the vulnerable code files from the benign ones. The efficiency of the developed models is evaluated on a publicly available labeled dataset that contains 9408 PHP labeled (i.e. safe, unsafe) source code files. The experimental results depict the superiority of the proposed approach over existing ones.

Keywords: Cross-Site Scripting vulnerability, Web Security, Vulnerability Detection, Machine Learning.
Communicated by: G-J Houben & Q. Li

1 Introduction

Nowadays, the usage of web applications is increasing very rapidly and has marked their presence in every sphere of our daily life [1]. At the same time, the security vulnerabilities have become a major concern to all. It has been noticed that application-level vulnerabilities are among the primary sources of Cybercrime. The exploitation of these vulnerabilities allow an attacker to breach the integrity, availability, and secrecy of the web applications, and resulting into some loss to the Internet users [2, 3].

[a]corresponding author

According to the WhiteHat Sentinel security testing report, approximately 56% of web applications have security vulnerabilities [4]. Among the variety of security vulnerabilities, XSS has been exhibited as the most critical vulnerability in web applications [5]. Authors of recent survey papers [3, 6, 7] have concluded that despite many solutions, the XSS remains a vital threat for web applications. They have urged for more research and real solutions for the development of XSS free web applications.

Researchers in paper [8, 9] have stated that the vulnerability prediction models play significant role in the detection of software vulnerabilities. The development of prediction model requires one important task i.e. feature extraction. Researchers have extracted a variety of features such as the line of code, code complexity, and other code attributes. The existing feature extraction approaches [10, 11, 12, 13] do not extract the features related to the context-sensitivity of a user-input. These features are essential for precise detection of XSS vulnerabilities (as explained in Section 2).

This paper proposes a novel approach for detecting vulnerable files in the web applications. It employs text-mining and pattern-matching techniques to extract two types of features i.e. *Basic features* and *HTML context features* from the source code files. The *Basic features* are the code attributes that represent input, output, sanitization, and string functions in the source code. And *HTML context features* represent the ways of referencing *user-input* in *output-statements*. These features are then used in the generation of a prediction model, which classifies unlabeled source code files into safe and unsafe categories.

To the extent of our knowledge, the proposed approach is first one that uses HTML context information in the building of prediction model. We have chosen to work with the PHP as it is the most commonly used language in web application development and prone to different types of security vulnerabilities [14].

The remainder of this paper is structured as follows. Section 2 outlines the motivation for the work. Section 3 summarizes the existing work in this area. The proposed vulnerability detection approach is explained in Section 4. Section 5 presents the experimental settings, dataset and performance measures. The results and observations are described in Section 6. Finally, the paper is concluded in Section 7.

2 Motivation

This section shows the presence of cross-site scripting vulnerabilities in the different HTML contexts through various code examples, and discusses the limitations of current vulnerability detection approaches.

2.1 Cross-Site Scripting Vulnerabilities

Cross Site Scripting is a weakness or loophole in the source code of web applications. It occurs when a user-input from an HTTP request, database, or any other files is used in an output-statement without proper validation, sanitization, or escaping process. It allows a malicious user to insert crafted scripting code into a vulnerable application that gets executed in the legitimate user's browser and leads to security threats [3].

Figure 1 shows a sequence of steps that can be used by an attacker to hijack the legitimate user's session.

It shows that first legitimate user is logged into a mail server, and then an attacker sends

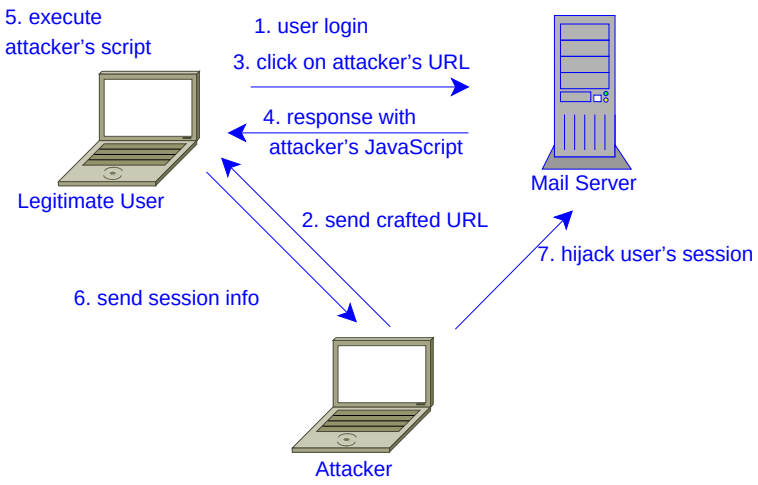

Fig. 1. sequence model for reflected XSS attack

a mail containing designed URL to the legitimate user (step 2). When the legitimate user clicks on (step 3) the created URL, the request goes to the mail server along with malicious scripts as parameter values. The mail server program processes the request and sends back the constructed script as a response to the legitimate user (step 4). The legitimate user's browser executes the script and sends victim's session information to the attacker (step 6). The attacker uses this information to hijack the legitimate user's session.

2.2 Limitations of Vulnerability Detection Approaches

In the review process, it has been identified that the majority of the current vulnerability detection approaches [15, 16, 17] consider the validation of a *user-input* by any standard sanitization routine such as *htmlspecialchars(), htmlentities* as an absence of XSS vulnerability. However, in current web applications, a user-input is referenced in an output-statement with HTML code to produce a dynamic HTML document. This combination represents a *HTML context* and enables the web browsers to interpret the HTML document differently. The common HTML contexts in which a *user-input* is referenced are HTML Element, Script, CSS, HTML Attribute, and URL context. Each context has different characteristics and requires different defense mechanisms to avoid XSS vulnerabilities [18].

For example, Listing 2.2 shows the code fragment of a vulnerable *search engine application* that accepts user input and returns the relevant results. In this example, user input (line 3) is used in an output statement (line 8) in the HTML element context.

Vulnerability in HTML element context

```
1 <html><body>
2 <?php
3  $mysearch= $_GET['myinput'];
4 // some database operations
5   $empty_results=1;
6   if ($empty_results)
7   {
```

```
8    echo "no result found for $mysearch";    }  ?>
9 </body></html>
```

A designed malicious search engine URL with attack payload

```
1 http://localhost/mycode/listing1.php?myinput=flower <script language= "JavaScript">
      document.location="http://localhost/mycode/stealer.php?cookie="+ %2B document.
      cookie; </script>
```

The exploitation of XSS vulnerabilities can be understood in the two steps. In the first step, an attacker designs a malicious URL for the search engine application (as shown in the Listing 2.2) and sends it to the legitimate user. This URL contains a variable *myinput* that default value is a designed scripting code used for stealing the user's cookie. In the second step, the legitimate user accesses his e-mail account and clicks on the attacker's designed URL. When the URL opens in the browser, the value of *myinput* is assigned in the *mysearch* variable of search engine (Listing 2.2, line 3) application. This value is used in the output statement (Listing 2.2, line 8) to generate the server response, which contains the designed script code. The execution of this code in the web browser transfers the user's cookie to the attacker's server. Further, the attacker can use these values to access the victim's mail account. The certain manipulations of IMG or IFRAME HTML tags allow the loading of the malicious code automatically in an HTML e-mail.

In our code testing experiments, it is found that a *htmlspecialchars()* is adequate to thwart the XSS attack in HTML element context (Listing 2.2). As it eliminates the consequence of special characters (e.g. $<, >$) but fall short in other contexts. To illustrate this, consider a *guestbook application* (in Listing 2.2) in which a user can write and display new messages with color formatting preference.

Vulnerability in HTML CSS context

```
1  <!-- server-side code fragment -->
2  <?php
3    $Color= $_GET['mycolor'];
4    $userName=$_GET['username'];
5    $message=$_GET ['msg'];
6    echo "<h1 style=\"color:$Color\">Welcome to $userName !!!</h1>";
7
8  // display a message
9    echo "<div style='background-color:$Color'>$message</div>";
10 ?>
11
12 <!-- Attack Vector 1
13 mycolor = red" > <script>alert("Attacked")</script> -->
14
15 <!-- Attack Vector 2
16 mycolor= green ' onmouseover=window.location='http://localhost/mycode/flash_movie_player.exe' '
       -->
```

Here, at the client-side, a user writes a message, selects the background color from the drop-down list, and then submits it to the server. At the server-side, the user's inputs are used in the output statements to generate an HTML document for displaying a message.

Listing 2.2 shows the server-side code fragment. In line 6 and 9, a user input, i.e., *mycolor* is used in the style attribute of an HTML tag to set the foreground and background color of text respectively. These lines represent a user-input in the CSS context. Suppose an attacker

prepares a URL using an attack vector 1 (shown in Listing 2.2 at line 13) as a value of *mycolor* and sends to a legitimate user. If the legitimate user clicks on the URL, then a simple pop-up message will get appeared. It shows a vulnerability in line 6, which allows an attacker to execute a malicious JavaScript code. This attack can also be prevented by using a standard sanitization function. However, the attack vector 2 (shown in Listing 2.2 at line 16) bypasses the standard sanitization functions and allows an attacker to perform malicious activity.

These examples conclude that the different defense mechanisms are required to avoid XSS vulnerabilities in different HTML contexts.

3 Related Work

This section presents a summary of current vulnerability detection and prediction approaches.

3.1 Vulnerability Detection Approaches

Huang et al. [19] were among pioneers to propose a taint analysis based approach for detecting the XSS vulnerabilities. Jovanovic et al. [20] proposed an approach based on the flow-sensitive and inter-procedural based analysis. They implemented first static source code analyzer tool, named *Pixy* [15] to detect many types of security vulnerabilities. In the review process, we identified that it detects XSS vulnerabilities with high false positive. Wassermann et al. [21] combined the taint analysis with string analysis to detect XSS vulnerability. However, their approach could not work effectively for HTML context-sensitive code. Shar et al. [18] employed a pattern-matching technique to handle the HTML context-sensitivity issue.

Most of these approaches rely on standard sanitization function and show the absence of XSS vulnerability if any standard function is used in the code, which is the main reason for high false results.

3.2 Vulnerability Prediction Approaches

Researchers in the papers [8, 9] discussed that the prediction models are used to detect vulnerable files in the code verification phase. Chowdhury et al. [9] proposed a framework to automatically predict vulnerability-prone files in the Mozilla Firefox application using cyclomatic complexity, cohesion, and coupling software metrics. They evaluated the performance of their prediction models built from four machine learning classifiers namely C4.5 Decision tree, Random Forest, Logistic Regression, and Naive Bayes on 52 releases of the Mozilla Firefox. They achieved an accuracy of 72% and recall values of 74% for C4.5 decision tree algorithm.

Shin et al. [8] used various matrices, i.e., code complexity, code churn, and developer activity to detect vulnerable source code files. They had built prediction model by using logistic regression machine learning algorithm and achieved an average recall of 80%. They examined that the complex programs are superfluous to vulnerability.

The above-discussed approaches showed that the probability of occurrence of vulnerabilities more in complex code. They used software metrics to build the prediction models by using different classification algorithm. On the other hand, authors of paper [22] stated that the utilization of an invalidated user-input is the primary source of XSS vulnerabilities. Shar et al. [10] showed that the small and uncomplicated program has many XSS vulnerabilities. Therefore, general vulnerability prediction models that use code metrics (such as code

complexity, and code churn) are not workable for detecting XSS vulnerabilities.

Hovsepyan et al. [23] employed a text-mining approach to extract a set of features and characterized each source code files using frequency of each feature. They applied this approach to detect vulnerable files in the source code of the software. The authors of paper [11] analyzed software metrics and text-mining based features and asserted the superiority of text-mining feature over the software-metrics feature in the detection of XSS vulnerabilities.

4 Proposed Vulnerability Detection Approach

Our goal is to develop a vulnerability prediction model for detecting the vulnerable files in web applications. A file is considered as vulnerable if at least one XSS vulnerability is present in it. It is defined in Eq. (1) as follows:

$$Vulnerable(file) = \begin{cases} yes, & \text{if number of XSS is } \geq 1 \\ no, & \text{otherwise} \end{cases} \tag{1}$$

The proposed XSS detection approach (as shown in Fig. 2) consists of two distinct phases - prediction model building and vulnerability detection phase. In the *prediction model building phase*, first, we prepare a training dataset of labeled source code files. Then, the basic and context features are extracted from each file in the dataset. The details of the proposed feature extraction algorithms are described in Subsection 4.1. Next, the joint feature sets are obtained corresponding to each file by combining their basic and context features. These feature sets are given to a machine-learning algorithm for the building of an XSS prediction model.

In the *detection phase*, the same procedure as used in the previous phase is applied to obtain feature sets from the test dataset, which consists of unlabeled source code files. These feature sets are given to the prediction model that detects vulnerable files in the test dataset.

4.1 Feature Extraction Algorithms

In this section, two algorithms for extracting the basic and context features from the given set of source code files are described. The algorithm 1 uses HTML DOM parser (`http://simplehtmldom.sourceforge.net/`) to determine a *block context* of each statement in a source code file. A *block context* of a statement is a name of HTML block in which it is present. In our approach, we have considered three block contexts namely Comment, Script, and Style. Except for the statements in these three contexts, all other statements are taken into account in HTML Body block context. The block contexts are very useful in the determination of two-level context sensitivity of a user-input in the output statements.

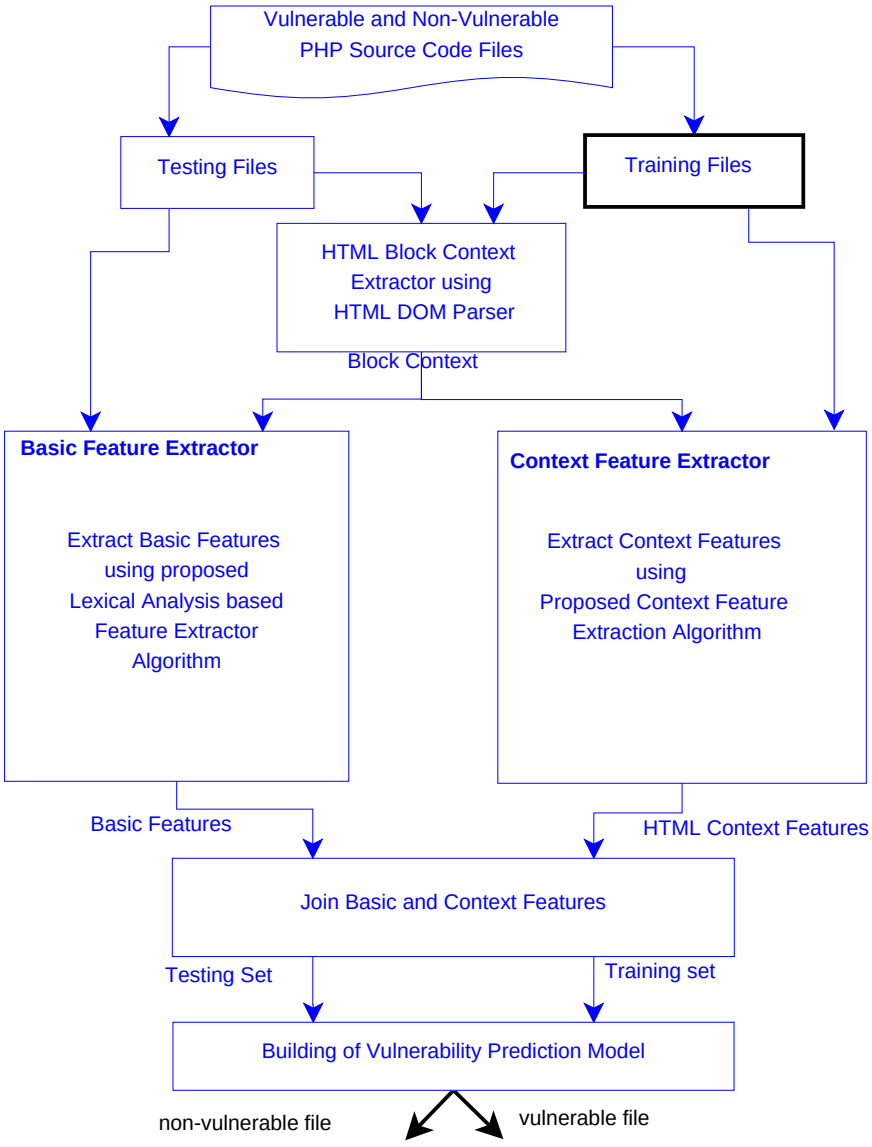

Fig. 2. flow diagram of vulnerability detection approach

Algorithm 1: Proposed Feature Extraction Approach

```
INPUT : Set of source code files S_Files
OUTPUT : Set of feature vectors S_fVector
N: Number of source code files
F_i: i^th source code file
S_{i,j}: j^th statement in F_i
BContext[]: An array of Block Contexts
C_block: Block Context
Token[]: An array of tokens
Global_var: A PHP global variable
FV_context : Context features t_k: k^th token
t_{k,name}: k^th token name
t_{k,val}: k^th token value
FV_i: i^th feature vector
tToken: Tagged token string
IToken[]: An array of ignorable tokens
T_tcs = T_CONSTANT_ENCAPSED_STRING
T_tew= T_ENCAPSED_AND_WHITESPACE
for each source code file F_i in S_Files do
    BContext[] = Extracted Block contexts for F_i
    for each C_block in BContext[] do
        for each statement S_{i,j} in C_block do
            Token[] = TokenGetAll (S_{i,j})
            for each token t_k present in Token[] do
                if (t_{k,name} is in IToken[]) then
                    No operation
                else if (t_{k,name}==T_ VARIABLE) then
                    if (t_{k,val} is in A_gVar) then
                        Add t_{k,val} in FV_i
                    else
                        tToken=concat(t_{k,name},C_block)
                        Add tToken in FV_i
                else if (t_{k,name} == T_ ECHO) then
                    tToken=concat(t_{k,name},C_block)
                    Add tToken in FV_i
                else if (t_{k,name}==T_tcs || t_{k,name}==T_tew) then
                    if (t_{k,val} contains HTML code) then
                        FV_context=Call Context_Finder(t_{k,val},C_block)
                        Add FV_context in FV_i
```

Next, each block statement is processed through a tailored tokenizing process, which generates a set of tokens. A token represents the language reserved words, strings, inputs, outputs, sanitization routine and other code attributes. The extracted block context is associated with statement's tokens, and these tagged tokens are included as features in our basic feature set. As mentioned earlier, a string in an output statement may represent an

HTML context; hence a separate processing is needed. Except for the tokens that are related to the strings, all other tokens name or value are included in our basic feature set.

Algorithm 2: Extraction of Context Features

```
INPUT : A String S and Block Context C_block
OUTPUT : Context features FV_context
C_user: user-input context in an output statement
DQ: Double Quote
SQ: Single Quote
NQ: No Quote
S: String
if (S contains a complete HTML Tag) then
    FV_context=C_block;
    return FV_context;
else if (S begin with < and end by=" | =' | =) then
    if (< follows any special tag (e.g. a|style|script) is in S) then
        if (event handler is present or not ) then
            C_user =C_block + "Event_Attr_Value";
            C_user= C_user+[DQ|SQ|NQ];
        else
            C_user =C_block + "STag_Attr_Value";
            C_user= C_user +(DQ|SQ|NQ);
    else if (event handler is in S) then
        C_user =C_block +"Tag_Event_Attr_Val";
        C_user= C_user+ (DQ|SQ|NQ) ;
    else if ( Is style keyword in string S) then
        C_user =C_block + "Tag_CSS_Attr_Value";
        C_user= C_user + (DQ|SQ|NQ);
    FV_context=C_user;
    return FV_context;
else if (Is S =="<Non_special_tag" ) then
    C_user =C_block + "Attr_Name";
    FV_context=C_user;
    return FV_context;
else if (Is S==" < " ) then
    C_user =C_block + Tag_Name;
    FV_context=C_user;
    return FV_context;
else
    Terms= A set of terms in the strings
    C_user =C_block + Term;
    add C_user in FV_context
    return FV_context
```

The Algorithm 2 is designed to process the strings. Firstly, it determines HTML contexts of user input. Then the extracted contexts are tagged with the block context and are included as the context feature in the feature set. If the defined condition does not match with a string that contains HTML code, then that string is further tokenized into a set of terms. These

terms are tagged with a block context and included as the context features in our feature set. Further, for each file, a set of unique features with their frequencies is determined and considered them into a joint feature set for the corresponding source code file. Finally, all joint feature sets are given to a machine-learning algorithm to build an XSS prediction model.

An extensive analysis of many source code programs was done, and efforts were made for extracting HTML contexts for all practically possible patterns. The proposed HTML context identification rules are working for all patterns except a situation, where PHP code is embedded into an HTML tag. The asymptotic complexity of context finding algorithm is O(1). Therefore, addition or revision of any rules requires the designing the new regular expressions for those patterns, which is not a very difficult task.

4.2 Example

We compare our approach with a text-mining based feature extraction approach by taking an example given in Listing 4.2. In this example, a user-input is utilized in output statements with different HTML contexts. The existing text-mining based approach (proposed by Walden et al. [12]) tokenizes a source code file and considers all the PHP tokens in it feature set. It does not deduce the context-sensitivity of a user-input and extracts the same set of features for the different output statements, which use user-input in the different HTML contexts.

HTML context-sensitive code statements

```
1  <?php
2  $input= $_GET['userData'];
3  echo "Simple Hello \n";
4  echo "No result for $input,try again    ";
5  echo "<span style=\"color:$input\"> Welcome </span>";
6  echo "<a href=\"$input\">login2</a>";
7  echo "<input name='slno' type='text' value='".htmlspecialchars($input)."'>";
8  echo "No result for  ".htmlspecialchars($input);
9  echo "<a href=".urlencode($input).">login1</a>";
10 echo "<input name='slno' type='text' value='".htmlspecialchars($input,ENT_QUOTES)."'>";
11 ?>
```

For example, the output statements (line 4, 5, and 6) include the user-input in the different contexts. However, they extract the same set of features. Authors of this approach have also considered the same feature i.e. T_STRING for different built-in functions (line 8, htmlspecialchars and line 9, urlencode), and parameters (ENT_QUOTES, line 10). However, the different built-in functions and their parameters can be used to avoid XSS attack in the different contexts. We have considered all these differences in the proposed feature extraction approach.

5 Experiments

The performance of the proposed approach is assessed with two approaches given by Hovsepyan et al. [23] and Walden et al. [12]. First, we build the distinct feature sets, namely uni-word feature set (F1), walden feature set (F2), and proposed feature set (F3), by applying Hovsepyan et al., Walden et al., and the proposed feature extraction approach respectively on the same dataset. Then, several XSS prediction models are built by using each of the feature sets with different machine-learning algorithms. In our experiments, a data-mining tool (WEKA) [24] with its default parameter setting is used for constructing and evaluating the

performance of the vulnerability prediction models.

5.1 Dataset Used

To evaluate the performance of different approaches, we use a publicly available dataset repository containing 9408 PHP source code files [26]. It has 5600 safe (i.e. non-vulnerable) and 3808 unsafe (i.e. vulnerable) code files. Evaluation of the different approaches is performed on this dataset that contains the variety of sample code files with their vulnerability labels.In addition to this, each code file has only one sensitive sink statement, which allows us to compare proposed approach with the statement level approaches.

The dataset is very suitable as compared to other repositories such as NVD (https://nvd.nist.gov/) and Bugzilla (https://www.bugzilla.org/) for evaluating the performance of different approaches. Because, these repositories provide only the vulnerability information and do not have the source code, which is required in our experiments. In addition, NIST (http://www.nist.gov/) also provides a dataset, but it has only 80 PHP source code files, which are insufficient to build an efficient prediction model.

5.2 Machine Learning Algorithms

A feature set with different machine learning algorithms generates the different prediction models and may produce different results. In our experiments, we have used seven machine-learning algorithms- Naive-Bayes (NB), Random Tree, Random forest, JRip, J48, Support Vector Machine (SVM), and Bagging - with the default parameter values to evaluate their prediction performance with the different feature sets. The details of these algorithms can be found in [25].

5.3 Experimental Setting

To evaluate the performance of different approaches a 10-fold cross-validation methodology is applied. In which, the chosen dataset is randomly divided into two disjoint training set and testing set containing 90% and 10% samples respectively. All the experiments are repeated ten times with a random selection of training and testing sets. Finally, the average results are accounting the concluding performance.

5.4 Performance Measures

The recall, precision, F-measures and accuracy measures are evaluated to conclude the efficiency of the prediction models. These measures can be described with the help of confusion metrics defined in Table 1, which shows the relationships between actual and prediction results as follows.

Table 1. Confusion Metrics

		Predicted	
		unsafe (vulnerable)	safe (non-vulnerable)
Actual	unsafe (vulnerable)	True Positive (TP)	False Negative (FN)
	safe (non-vulnerable)	False Positive (FP)	True Negative (TN)

1. **Recall:** Recall is defined in Eq. (2) as a fraction of the number of accurately determined unsafe files to the actual number of unsafe files. It measures a vulnerable file detection rate.

$$Recall = \frac{TP}{(TP + FN)} \tag{2}$$

2. **Precision:** Precision is defined in Eq. (3) as a fraction of accurately determined unsafe files to total number files which are predicted as unsafe. It measures the correctness of predictor to identify unsafe files.

$$Precision = \frac{TP}{(TP + FP)} \tag{3}$$

   The value of precision and recall parameters must be high for an efficient machine learning model.

3. **Accuracy:** Accuracy is defined in Eq. (4) as a fraction of accurately determined safe or unsafe files to total tested files (i.e. safe or unsafe). It represents the correct classification rate of the predictor.

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{4}$$

4. **F-Measure:** it is defined in Eq. (5) as a weighted average of precision and recall:

$$F_B measure = (1 + B^2)\frac{(Precision * recall)}{((B^2 * precision) + Recall)} * 100 \tag{5}$$

   Here B represents relative weight of recall and precision. In our evaluation, we have used F1 measures in which recall weights is equal to the precision.

6 Results and Discussion

Various experiments are performed to compare the predictor's performance in many ways. From our experiments, it is observed that the proposed feature set produces the highest performance as compared to the feature sets of the existing approaches.

6.1 Prediction Performance for Vulnerable Class Files

Our objective is to correctly detect the most of the vulnerability-prone files with minimum false positive and negative results. Many prediction models based on proposed features and different Machine-Learning(ML) algorithms are developed for evaluating prediction performances. Table 2 shows the results of different predictors performance measures for vulnerable class files. As mentioned earlier, recall and precision value must be high for an efficient prediction model.

Table 2. Vulnerability prediction performance for vulnerable class files

Machine-Learning Algorithm	**Precision**	**Recall**	**F-Measure**	**Accuracy**
NB	67.77	47.01	55.5	69.5
SVM	89.34	82.69	85.88	88.99
Bagging	93.59	86.87	90.09	92.6
JRip	99.29	61.48	75.53	83.6
Random Forest	82.52	83.1	82.8	87.62
J48	93.31	86.59	89.8	92.04
Random Tree	77.23	79.27	78.22	82.12

For recall values, bagging based model outperforms all other models. For example, the bagging algorithm recall is 86.87%, which is the best among all the other algorithm's recall, i.e., 47.01%, 82.69%, 61.48%, 83.1%, 86.59%, and 79.27% for NB, SVM, JRip, Random Forest, J48, and Random Tree algorithms respectively. In contrast, many vulnerable-prone files may remain undetected if a prediction is made using a NB based model. It is also observed that there is not a significant variation between recall results of SVM and Random Forest algorithms.

Considering precision as a performance measure, JRip based model produces the highest precision, i.e., 99.29%. It indicates that JRip does not produce false positive results (non-vulnerable as vulnerable). However, JRips recall, i.e., 61.48% indicates that it does not detect all the vulnerable files correctly and report some of them as non-vulnerable (false negative), which is not accepted for a good vulnerability prediction model.

For a more explicit representation,the recall, precision and F-measure for different prediction models is shown Fig. 3. It represents that there is a trade-off between precision and recall. As already mentioned, the sole consideration of recall or precision may be misleading. Therefore, the performance of any model can be best justify by using F-measure. In our experiments, it is observed that bagging based model gives the highest recall as well as F-measure as compared to other predictors. (shown in Fig. 3).

6.2 Comparison with Current Related Approaches

This section compares the results of the proposed approach with related state-of-art text-mining approaches on the same dataset. Hovsepyan et al. [23] proposed a text-mining based prediction model for predicting vulnerable files in the source code of software application. They treated *unique words* in source code files as independent variables. Walden et al. [12] assumed all unique PHP tokens in source code files as independent variables and employed only the random forest algorithm to build a prediction model. In this paper, three feature

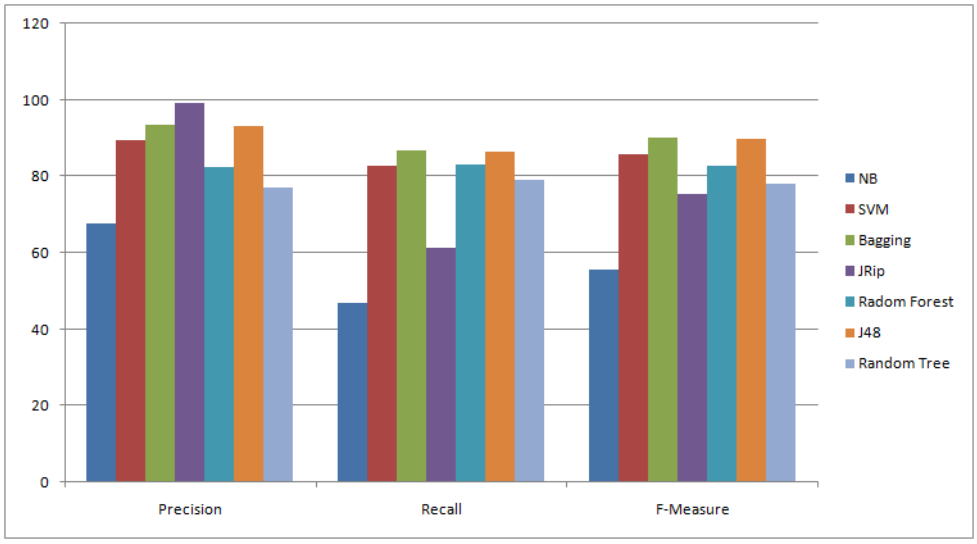

Fig. 3. precision v/s recall of different prediction models

sets are extracted by applying different approaches on the same dataset.

Table 3 shows the weighted average results of three approaches with various performance measures- precision, recall, accuracy, and F-measure.

Table 3. Weighted average performance measures for prediction models

ML Algo	**uni-word features (F1), unique-token features (F2), proposed features (F3)**											
	Precision			**Recall**			**F-Measure**			**Accuracy**		
	F1	F2	F3‘	F1	F2	F3	F1	F2	F3	F1	F2	F3
NB	61.6	63.7	69.2	61.4	64.7	69.5	61.5	62.9	68.2	61.4	64.6	69.5
SVM	68.8	70.5	88.9	68.7	70.9	88.9	68.7	70.5	88.9	68.7	70.9	88.9
Bagging	69.4	70.9	92.7	69.3	71.3	92.6	69.2	70.8	92.6	69.3	71.2	92.6
Random Forest	67.9	68.9	87.6	67.6	69.4	87.6	67.8	68.8	87.5	67.6	69.4	87.6
J48	70.3	71.2	92	70.1	71.6	92	70.2	71.1	91.9	70.1	71.6	92
JRip	65.5	69.3	86.9	65.3	69.7	83.6	65.4	69.1	82.6	69.7	69.7	83.6

It depicts that the proposed feature set produces the best accuracy, as high as, 92.6%, which is significantly higher to 69.3% and 71.2% for the uniword (F1) and uni-token(F2) feature sets, respectively, with the bagging machine-learning algorithm. From the Table 3, it is observed that the proposed feature set also produces better results as compared to other considered feature sets with the other machine-learning algorithms. The proposed approach outperforms the other approaches because it uses HTML context of user input in determining the feature set, while others have not.

From the Table 3, it can be seen that the SVM algorithm confer superior results in comparison to NB, random forest and JRip algorithms with different feature sets. It also depicts that the bagging algorithm results are very close to the J48 algorithm results in the

different experiments. Further, we also found that the uni-word(F1) and unique-token(F2) feature set produces the higher performance with J48 machine-learning algorithm as compared to the other considered algorithms. For example, the uni-token feature (F2) set produces an accuracy of 71.6%, which is the best among all the other algorithm's accuracy i.e. 64.6%, 70.9%, 71.2%, 69.4%, and 69.7% for NB, SVM, bagging, Random Forest, and JRip algorithms respectively.

6.3 Performance of Different Machine-Learning Algorithms

This subsection presents the results of a statistical significance test, which is used to determine whether the difference in performance measures for different prediction models (i.e. predictors) are statically significant or not. It provides a pair-wise comparison of predictors using a corrected standard t-test. The Random Forest based predictor is considered as a baseline to perform a standard t-test at 0.05 significance. Because in the literature, a corrected standard t-test at a significance level of 0.05 or less than is considered statistically significant [9].

The results of mean and the standard deviation in accuracy for different machine-learning algorithms are shown in Table 4. It depicts the statistical significance test results for accuracy performance measures. We have used "Yes+", "Yes-" or "No" annotations to represent the statistical test results. A statistically better or worse result from the baseline predictor result is represented by the "Yes+" or "Yes-" annotations respectively. On the other hand, when the value of two results are different and a difference in the result is statistically insignificant, then it is represented by the "No" annotation.

Table 4. Prediction accuracy, standard deviation and T- test results

Machine-Learning Algorithm	**Mean Accuracy**	**Standard Deviation**	**T-Test Result**
Random Forest	87.62	1.39	
SVM	88.99	1.44	Yes+
Bagging	92.6	1.47	Yes+
JRip	83.6	1.98	Yes-
NB	69.5	1.8	No
J48	92.04	1.81	Yes+
Random Tree	82.12	1.83	Yes-

From the Table 4, we can infer many points. First, the all prediction models accuracy is more than 80%. It shows the usefulness of the proposed features in the building of a vulnerability prediction model. Second, the standard deviations in accuracies for different ML algorithms are very low, which shows that there is low variation in accuracy for different training and testing sets in cross-validation experiments. Third, the bagging algorithm performance is better than the other considered algorithms and it is very close to J48 algorithm performance on the same dataset. For example, the bagging algorithm produces an accuracy of 92.6%, which is the best among all the other algorithm's accuracy i.e. 69.5%, 82.12%, 83.6%, 87.62%, 88.99% and 92.04% for NB, Random Tree, JRip, Random Forest, SVM and J48 algorithms respectively.

7 Conclusion and Future work

This paper proposed a novel feature extraction approach to extracts the basic as well as context features from source code files. The experimental results have shown that the proposed approach produced the highest accuracy as compared to the existing feature extraction approaches in the detection of XSS vulnerabilities. The reason for the superiority of proposed approach can be attributed to the consideration of the context-sensitivity of user input in determining feature set. The proposed approach can assist the web software engineers to detect the probable vulnerable code files during development life cycle and save their time by focusing more on only those files to mitigate the vulnerabilities.

The certain limitations of the approach and the results are as follows: First, the proposed approach could be used to detect reflected and stored XSS vulnerability-prone files and does not work for DOM XSS vulnerability-prone files. Second, in our experimental work, all the performance measures results are obtained using default parameter setting of the machine-learning algorithms. Though, the efficiency of these models may vary by changing the parameters values. However, we have not attempted to identify the most efficient parameter settings of the prediction models. As our objective is to determine the usefulness of proposed features in the vulnerability-detection task, which are already justified in the results and discussion section. In future, the same approach will be exercised to analyze SQL Injection vulnerabilities.

References

1. Shashank Gupta and B.B. Gupta (2016), *XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code*, Arabian Journal for Science and Engineering, Vol. 41(3), pp. 897–920.
2. Divya Rishi Sahu and Deepak Singh Tomar (2017), *Analysis of Web Application Code Vulnerabilities using Secure Coding Standards*, Arabian Journal for Science and Engineering, Vol. 42, pp. 885-895.
3. Isatou Hydara and Abu Bakar Md. Sultan and Hazura Zulzalil and Novia Admodisastro (2015), *Current state of research on cross-site scripting A systematic literature review*, Information and Software Technology, Vol. 58, pp. 170 - 186.
4. *WhiteHat Security Statistics Report*, https://www.whitehatsec.com/categories/statistics-report/, Accessed: 2015-09-21
5. *Open Web Application Security Project*, https://www.owasp.org/index.php/Top_10_2013-Top_10, Accessed: 2016-06-26
6. G. Deepa and P. Santh Thilagam (2016), *Securing Web Applications from Injection and Logic Vulnerabilities*, Inf. Softw. Technol., Vol. 74, pp. 160–180.
7. D.B. Lowe and J. Eklund (2016), *Web application protection techniques: A taxonomy*, Journal of Network and Computer Applications, Vol. 60, pp. 95-112.
8. Yonghee Shin, A. Meneely, L.Williams, and J.A. Osborne (2011), *Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities*, IEEE Transactions on Software Engineering, Vol. 37(6), pp. 772-787.
9. Istehad Chowdhury and Mohammad Zulkernine (2011), *Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities*, Journal of Systems Architecture, Vol. 57(3), pp. 294 - 313.
10. Lwin Khin Shar and Hee Beng Kuan Tan (2013), *Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns*, Information and Software Technology, Vol. 55(10), pp. 1767 - 1780.

11. Riccardo Scandariato and James Walden and Aram Hovsepyan and Wouter Joosen (2014), *Predicting Vulnerable Components: Software Metrics vs Text Mining*, IEEE 25th International Symposium on Software Reliability Engineering (ISSRE), pp.23-33.
12. R. Scandariato and J. Walden and A. Hovsepyan and W. Joosen (2014), *Predicting Vulnerable Software Components via Text Mining*, IEEE Transactions on Software Engineering, Vol.40(10), pp. 993-1006.
13. Lwin Khin Shar and Hee Beng Kuan Tan and Lionel C. Briand, (2013), *Mining SQL Injection and Cross Site Scripting Vulnerabilities Using Hybrid Program Analysis*, Proceedings of the 2013 International Conference on Software Engineering(ICSE '13), pp. 642 - 651.
14. *Extensive and reliable web technology surveys*, http://w3techs.com/technologies/overview/programming_language/all, Accessed: 2015-09-10
15. Nenad Jovanovic and Christopher Kruegel and Engin Kirda (2006), *Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)*, Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 258-263.
16. Ibéria Medeiros and Nuno F. Neves and Miguel Correia, (2014), *Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives*, Proceedings of the 23rd International Conference on World Wide Web (WWW '14), pp. 63-74.
17. Johannes Dahse, *Static Source Code Vulnerability Analyzer*, http://rips-scanner.sourceforge.net/, Accessed: 2016-07-13
18. Lwin Khin Shar and Hee Beng Kuan Tan (2012), *Automated removal of cross site scripting vulnerabilities in web applications*, Information and Software Technology, Vol. 54, pp. 467-478.
19. Yao-Wen Huang and Fang Yu and Christian Hang and Chung-Hung Tsai and Der-Tsai Lee and Sy-Yen Kuo (2004), *Securing Web Application Code by Static Analysis and Runtime Protection*, Proceedings of the 13th International Conference on World Wide Web (WWW '04), pp. 40–52.
20. Nenad Jovanovic and Christopher Kruegel and Engin Kirda (2006), *Precise Alias Analysis for Static Detection of Web Application Vulnerabilities*, Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security (PLAS '06), pp. 27-36.
21. G. Wassermann and Su Zhendong (2008), *Static detection of cross-site scripting vulnerabilities*, ACM/IEEE 30th International Conference on Software Engineering(ICSE '08), pp. 171-180.
22. Lwin Khin Shar and Hee Beng Kuan Tan (2012), *Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns*, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), pp. 310-313.
23. Aram Hovsepyan and Riccardo Scandariato, and Wouter Joosen and James Walden, (2012), *Software Vulnerability Prediction Using Text Analysis Techniques*, Proceedings of the 4th International Workshop on Security Measurements and Metrics (MetriSec '12), pp. 7-10.
24. Eibe Frank and Mark Hall and Peter Reutemann and Len Trigg, *WEKA: Data Mining Tool*, http://www.cs.waikato.ac.nz/ml/weka, Accessed: 2016-06-26
25. Ian H. Witten and Eibe Frank and Mark A. Hall (2011), *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann Publishers Inc.
26. Aurelien DELAITRE, Bertrand STIVALET, *PHP Vulnerabilities Test Suite*, https://github.com/stivalet/PHP-Vulnerability-test-suite, Accessed: 2014-07-13
27. Prateek Saxena and David Molnar and Benjamin Livshits, (2011), *SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications*, Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11), pp. 601–614.
28. Peng Li and Baojiang Cui (2010), *A comparative study on software vulnerability static analysis techniques and tools*, IEEE International Conference on Information Theory and Information Security (ICITIS), pp. 521-524.
29. G. Agosta and A. Barenghi and A. Parata and G. Pelosi (2012), *Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution*, Ninth International Conference on Information Technology: New Generations (ITNG), pp. 189-194.
30. *Common Vulnerabilities and Exposures*, https://cve.mitre.org/
31. *XSS Attack Information*, http://www.xssed.com/, Accessed: 2015-09-20