An Unsupervised Method for Detection of XSS Attack

Swaswati Goswami[1], Nazrul Hoque[1], Dhruba K. Bhattacharyya[1], Jugal Kalita[2]
(Corresponding Author: Dhruba K. Bhattacharyya)

Department of Computer Science and Engineering & Tezpur University[1]
Napaam, Sonitpur, Assam-784028, India
(Email: dkb@tezu.ernet.in)
Department of Computer Science, University of Colorado[2]
Colorado Springs, O 80933-7150, USA[2]

(Received Jan. 3, 2016; revised and accepted May 20 & Apr. 9, 2016)

Abstract

Cross-site scripting (XSS) is a code injection attack that allows an attacker to execute malicious script in another user's browser. Once the attacker gains control over the Website vulnerable to XSS attack, it can perform actions like cookie-stealing, malware-spreading, session-hijacking and malicious redirection. Malicious JavaScripts are the most conventional ways of performing XSS attacks. Although several approaches have been proposed, XSS is still a live problem since it is very easy to implement, but difficult to detect. In this paper, we propose an effective approach for XSS attack detection. Our method focuses on balancing the load between client and the server. Our method performs an initial checking in the client side for vulnerability using divergence measure. If the suspicion level exceeds beyond a threshold value, then the request is discarded. Otherwise, it is forwarded to the proxy for further processing. In our approach we introduce an attribute clustering method supported by rank aggregation technique to detect confounded JavaScripts. The approach is validated using real life data.

Keywords: Attribute Clustering; Divergence; Malicious Script; Proxy; XSS

1 Introduction

Cross-site Scripting (XSS) is one of the most common application layer hacking techniques. It allows an attacker to embed malicious JavaScript, VBScript, ActiveX, HTML or Flash into a vulnerable dynamic page to fool the user, executing the script on his/her machine in order to gather data [17]. Most common way of stealing cookies or hijacking session is to embed a JavaScript encoded with browser supported HTML encoding technique. XSS attacks are categorized into three types [10]: reflected XSS, stored XSS and Document Object Model or DOM-based XSS attack.

As the Internet applications are becoming more and more dynamic, the possibilities of such attacks have become more prominent. The number of vectors which are used to carry out such attacks are increasing with the increase in interactiveness of an application. Severeness of XSS attack can easily be predicted as it is ranked in top positions in recent security related surveys. For example, XSS is ranked third in the "OWASP Top 10 Application Security Risks-2013" [32].

Most of the existing intrusion detection systems which are designed to detect the XSS attack consider that, XSS attack is substantially caused by the failure of a Web application to check the contents for malicious codes before running it in the user's browser. The existing approaches can be categorized into three basic types [27]: dynamic approach, static approach, and hybrid approach. Static analysis includes various methods such as taint propagation analysis [20], string analysis [31], software testing techniques [28], etc. Taint propagation analysis includes construction of a control flow graph, where each node contains a label. An Web page is considered vulnerable, if the input node of the control flow graph for a certain variable has an edge leading to the output node. In string analysis, the program generated string values contain formal language expressions, such as Context Free Grammar (CFG) with labels. Minamide's method [24] of string tainting, which is an example of string analysis approximates string output of a program with a CFG . Software based testing techniques such as fault injection, penetration testing are used to infer the existence of vulnerabilities. Dynamic analysis includes proxy based solutions [21], browser enforced embedded policies [19], etc. In proxy based solutions, requests from the client side are intercepted in the proxy and based on the rules of the proxy the required actions are taken. On the other hand, in browser enforced embedded policies client is provided with a list of benign scripts by the Web application and only these scripts are run. Although the static and dynamic solutions are effective in various cases, but in some situations the combination of both the approaches is much needed. Hence, the hybrid approaches are introduced. Sanar [3] is such a tool which combines static and dynamic approaches. In static analysis, it models the data input methods to indicate sanitization process. On the other hand, the code irresponsible for sanitization is reconstructed by dynamic analysis approach.

Machine learning based approaches use statically and dynamically extracted characteristic features from both malicious and benign samples and build classification tools [4].

1.1 Motivation

Although several methods have been introduced so far to mitigate XSS attack, it is still a live problem. The attack instances are increasing continuously and intruders are introducing more complex ways for embedding their scripts to trick users. Motivation behind choosing JavaScripts is that, now a days most of the web applications use JavaScripts extensively and the XSS attacks reported so far are in maximum cases found to be executed using JavaScripts. Moreover, already existing incremental approaches show a high false alarm rate and are not scalable [5]. Taking the whole scenario into consideration we are motivated to introduce a faster, stable and cost effective detection mechanism which will ensure high detection accuracy. Our aim is to reduce the false alarm rate and to increase the scalability.

1.2 Contributions

The two major contributions of this work are:

- A load balanced Client-Server based architecture to support XSS attack detection.

- An attribute clustering technique to support feature-level unsupervised grouping of attack and normal scripts over relevant and optimal feature space.

The remainder of this paper is organized as follows. The background of XSS attacks and a brief discussion on why we have concentrated mainly on reflected XSS attack is discussed in Section 2. Section 3 gives an overview of related works. Section 4 introduces our proposed method which is followed by experimental results in Section 5. Finally, we conclude our paper in Section 6.

2 Background and Related Work

In this section, we discuss the basics of XSS attacks, their categories and characteristics.

2.1 Basics of XSS Attacks

XSS attack mainly occurs due to the improper sanitization and validation of the user inputs given in the form of scripts. Figure 1 depicts a typical scenario of XSS attack. It shows, how an attacker can easily generate an XSS attack by sending a mail to the user containing a malicious URL. In the first step, attacker crafts a URL containing the malicious script and e-mail it to the victim. In Step 2, user clicks on the link send by the attacker and on clicking the link, the script is sent to the web server as the user request as shown in Step 3. In Step 4, the server reflects back the request to the user and the script is executed in the user's browser. Once the script is executed, sensitive data like session cookies are sent to the attacker in Step 6. Then attacker gets control over the user's session and can access the Web server on behalf of the user. The methods through which one can execute an XSS attack can be categorized into the following three types.

2.1.1 Persistent or Stored XSS Attack

Persistent or Stored XSS attack is server database related and it can affect a numerous number of users visiting the server which contains the malicious script in its database injected by the attacker. This type of attack mainly occurs due to the improper validation of the user inputs. Let us take an example to clarify the statement. A guest-book, which is a visitors log through which they can post their query or just leave a comment or give feedback for some services provided by a Website can be an easy victim of persistent XSS attack. Suppose a malicious user crafts a special script for cookie stealing and posted that as a comment in the guest-book. This malicious link may be a link to provoke the user for getting free recharge by posting a link with the tagline *"Hey check this link. I got free Recharge!!!"*. If the server is not able to sanitize the input properly then this comment is saved to the server database. Now the visitors visiting that particular Webpage will execute that javascript in their browser unknowingly. The attacker will thus get the cookies of the user's browser and thus will get the control over the user's session.

2.1.2 Non-Persistent or Reflected XSS Attack

Recently, non-persistent or reflected XSS attacks have been found as a common type of XSS attack. Here, victim's request itself contains the malicious string. The server then responds with an HTML page that contains the script and thus the script is executed in the user's browser. Let us consider the following scenarios.

Scenario 1: Email is one of the most common ways of tricking a user to click on a malicious script. The attacker can send a link to the user via an email crafting a link which contains the malicious link. As and when the link is clicked by the user, the script which is hidden either in the link itself or in a script

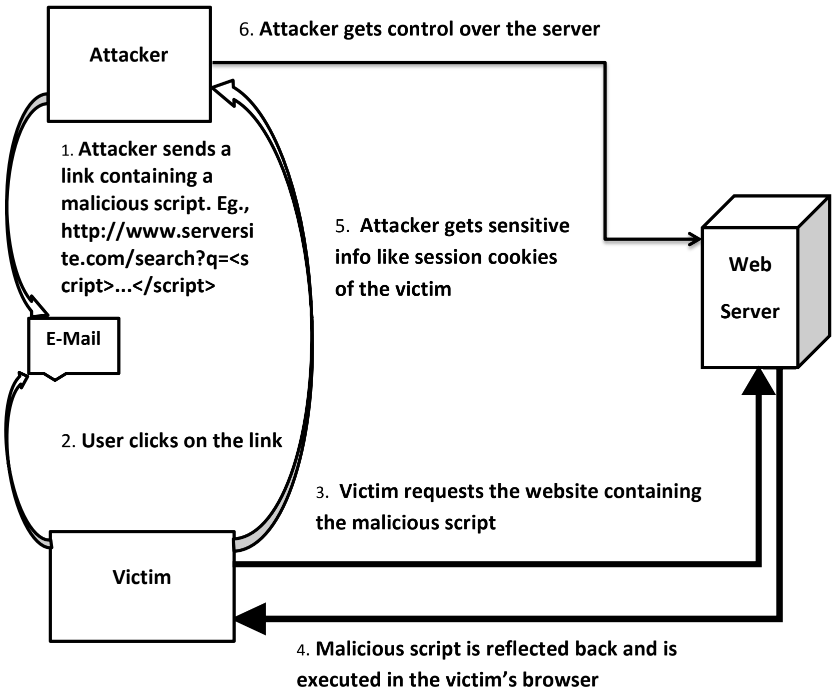

Attacker
6. Attacker gets control over the server
1. Attacker sends a link containing a malicious script. Eg., http://www.serversite.com/search?q=<script>...</script>
5. Attacker gets sensitive info like session cookies of the victim
Web Server
E-Mail
2. User clicks on the link
3. Victim requests the website containing the malicious script
Victim
4. Malicious script is reflected back and is executed in the victim's browser

Figure 1: An overview of XSS attack

which the attacker refers to is executed and users credentials such as session cookies. are sent to the attacker. The attacker can easily get access to the site which the legitimate user is surfing. Figure 2(a) shows the scenario of this attack.

Scenario 2: We can consider yet another scenario of XSS attack. Here, the attacker acts as an intermediary agent. The attacker can be the host of a legitimate Website. When the user visits the attacker's Website, then the attacker prompts with a specially crafted link. When the user clicks on the link, it redirects the user to another Website to which the user have access to. This reflected message can contain a script, which is then executed in the user's browser and the attacker can get the browser info this way. The link may contain a page which actually doesn't exist on the requested server. Then the server sends back a message to the user saying that *page not found*. This scenario is depicted in Figure 2(b).

2.1.3 DOM-based XSS Attack

DOM-based XSS attack is the type of XSS attack that occurs in the Document Object Model (DOM) of an HTML page in lieu of the part of an HTML page. Here, since the changes occur to the DOM environment, so the HTTP response code runs in a different manner. DOM XSS attack can be carried out with a numerous DOM objects as mentioned below.

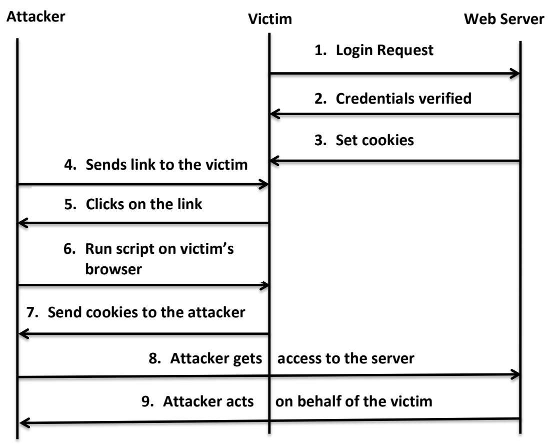
- User name or password part of a location or URL: Here the payload is received by the server in the authentication header.
- Portion where the query part is located in the URL: Here the payload is received by the server as URL part of HTTP request.
- Fragment part of an URL: This part basically contains the portion of the URL separated by '#' symbol from the rest of the URL. Here payload is not received by the server.
- HTML DOM referrer object: The referrer object is the *document.referrer*, which represents currently loaded document's URL. Here the payload is received by the server at the referrer header.

A report by Trustwave's Spiderlabs says that the number of applications that are vulnerable to XSS attack are 82% of the total Web applications (2013)[1]. Again, according to WhiteHat Security XSS stood first in the most common vulnerability category (2014)[2]. XSS also tops the list of most frequently occurring vulnerability in the survey carried out by Cenzic (2014)[3]. CWE by MITRE [8] also warns by saying that XSS is one of the most prevalent, obstinate and dangerous vulnerability in Web application. Among the XSS vulnerabilities, the most frequent one is the reflected XSS attack.

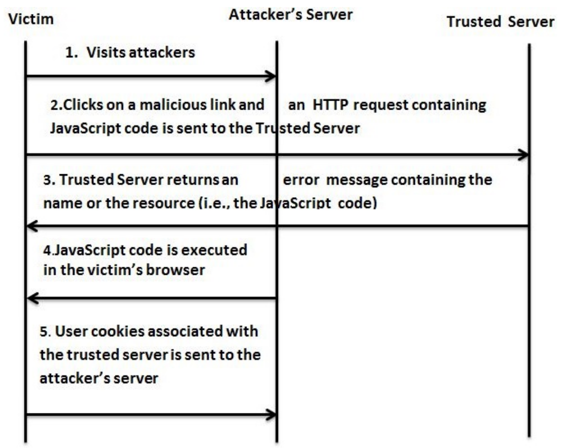
[1]https://www.trustwave.com/Resources/Library/Documents/2013-Trustwave-Global-Security-Report/

[2]http://info.whitehatsec.com/rs/whitehatsecurity/images/statsreport2014-20140410.pdf

[3]http://info.cenzic.com/rs/cenzic/images/Cenzic-Application-Vulnerability-Trends-Report-2013.pdf


Attacker
Victim
Web Server
1. Login Request
2. Credentials verified
3. Set cookies
4. Sends link to the victim
5. Clicks on the link
6. Run script on victim's browser
7. Send cookies to the attacker
8. Attacker gets access to the server
9. Attacker acts on behalf of the victim

(a) Reflected XSS attack generated through E-mail


Victim
Attacker's Server
Trusted Server
1. Visits attackers
2.Clicks on a malicious link and an HTTP request containing JavaScript code is sent to the Trusted Server
3. Trusted Server returns an error message containing the name or the resource (i.e., the JavaScript code)
4.JavaScript code is executed in the victim's browser
5. User cookies associated with the trusted server is sent to the attacker's server

(b) Reflected XSS attack generated through the links on malicious user's server

Figure 2: Different scenarios to generate reflected XSS attacks

In this work, we consider the attack instances of reflected XSS attacks. Reflected or Non-persistent XSS attack is the most common attack among all the three XSS attacks. It is easy to create and also can be launched easily to gather sensitive informations. So, attackers generally tend to carry out such attacks frequently. Since the Web service users are the common people who may not have an insight knowledge of the underlying architecture. So it becomes easy for the attackers to trick such individuals by creating a specially crafted URL and making the user to click on that. Moreover, since here the attacker crafted script is reflected back to the user's browser, so one need not to store the script in the server and to wait for user to check that Website for executing the attack.

2.2 Related Work

Javascript has become an unavoidable part of most Web applications due to the necessity of increased interactivity between the user and the Web applications. So the idea of detection of malicious JavaScript is not new. A brief summary of the work done so far, that are thoroughly studied to understand the present scenario in this field, are mentioned in Table 1.

In [22], Likarish et al. propose a classification based approach for detecting obfuscated malicious JavaScript detection. They propose features that can identify obfuscation, since obfuscation is a well known method of bypassing security filters. Based on the recommendation of various clssifiers, the designed malicious JavaScript detector either passes or discards the request.

In [21], Kirda et al. propose a tool *Noxes*. This is a client side solution to mitigate XSS-attack. It uses both manual and auto generated rules to mitigate XSS-attacks. Since we are focusing on designing a solution which balances the loads among server and the client, so it is very important for us to study the existing client based and server based approaches. Noxes identifies all the links either as statically embedded or dynamic links. Dynamic links are considered vulnerable to XSS attack, since attacker can embed their code in a dynamic link.

In [33], Wurzinger et al. propose a tool *Secure Web Application Proxy(SWAP)*, which is a server-side solution for mitigating XSS attack. SWAP consists of a reverse proxy. It interprets the HTML responses and the modified web browser detects the script contents.

In [23], Di Lucca el al. propose an approach, which is a combination of both static and dynamic approach. Static analysis is used to determine whether a server page is vulnerable to XSS attack. Dynamic approach verifies whether the determined vulnerable web application by static approach is actually vulnerable or not. This approach uses a control flow graph (CFG) to determine the vulnerability in a Web application.

In [28], Salas et al. propose an approach which uses security testing methods like penetration testing and fault injection for detection of XSS attack. Depending upon the results of penetration testing by a user utility referred to as soapUI and interpretation of HTTP status codes in the header of SOAP message, they develop 8 rules. On the basis of which they determine the existence of vulnerability in Web services. Fault injection phase is carried out with WSInject, which is placed as proxy between client and server and intercept the messages sent by soapUI before passing it to the server. Faults are injected during this phase. By intercepting the HTTP messages sent by the SOAP request message, they use the previously defined vulnerability analysis rules to determine the injection.

In [2], Athanasopoulos et al. present a tool called *xHunter*, which checks the JavaScript parse tree depth. If the depth is beyond some threshold value, it considers the URL as suspicious.

In [1], Adi et al. propose a design for a proxy named *Wines* that monitors the browser requests sent to a server. Depending upon the patterns of malicious strings kept in different cells of $Wines(T_H1, T_H2)$, the requests are categorized as either harmful or harmless. Harmless strings are forwarded to the server and harmful strings are blocked. All the terms used by the method are biological term since the work is inspired by Human Immune

Table 1: Comparison among existing methods

Referred Work	Year	Description	Dataset(R/S)
Likarish el al. [22]	2009	This technique propose a method to suppress potentially malicious JavaScripts based on the recommendation of classifiers.	S
Kirda el al. [21]	2006	This is a rule based client side solution to mitigate XSS attack.	R
Wurzinger el al. [33]	2009	This server side solution intercepts all HTML responses, and uses a modified Web browser which is utilized to detect script content.	R
Di Lucca el al. [23]	2004	This approach is a combination of static and dynamic approach for detecting XSS attack.	R
Salas et al. [28]	2014	This method is to analyze the robustness of web services by fault injection with WSInject.	R
Athanasopoulos et al. [2]	2010	Proposes a method called xHunter to detect XSS exploits from web trace.	R
Shar et al. [29]	2013	Hybrid model for XSS and SQL injection attack detection	S
Adi et al. [1]	2012	Proposes a method called Wines to detect mutated attack strings.	R
Gupta et al. [11]	2015	Proposed a method to prevent XSS attacks using Apache Tomcat and Web Goat	S
Chun et al. [9]	2016	XSS Attack Detection Method based on Skip List	S

R=Real life, S=Synthetic

System.

Gupta et al. [12] proposed a method called XSS-SAFE for XSS attack detection and prevention based on automated feature injection statements and placement of sanitizers in the injected code of JavaScrip. The main advantage of this method is that it can detect XSS attacks without any modification to client- and server-side commodities.

Our approach is a Client-Server based approach, which focuses on balancing the load between client and the server. The detection mechanism in the proxy includes an ensemble based feature selection approach followed by an attribute clustering method to distinguish the malicious traffic from the benign traffic.

3 Proposed Method

The following definitions and theorem provide the theoretical basis of our work. The symbols/notations used to describe our work are reported in Table 2.

Definition 1. ***Attribute Rank:*** *The rank of an attribute D_{a_i} is defined as the relevance of the attribute a_i for a given class (attack or normal) in a dataset D.*

Definition 2. ***Attribute Cluster:*** *An attribute cluster C_k^i of an attribute D_{a_i} is defined as a subset of objects of a given dataset D (i.e., $C_k^i \subseteq D_{a_i}$) which has high intra-cluster similarity over the attribute D_{a_i}.*

Definition 3. ***Cluster:*** *A cluster C_A is a subset of objects of a given dataset D (i.e., $C_A \subseteq D$) which is obtained by considering the common objects C_A^i over a selected subset of relevant attributes S. In other words,*

$$C_A = C_A^1 \cap C_A^2 \cap ...C_A^S, \quad S \leq n$$

Theorem 1. *If SD_attrib_clus() assigns an instance/object O_j to attack group C_A, it cannot be in the normal group of a relevant attribute cluster, w.r.t predefined attribute rank or relevance, i.e., $O_j \notin C_N^i, \forall i = 1, ..., S$.*

Proof: *It can be proved by contradiction.*
Let an object $O_j \in C_A$ as given by SD_attrib_clus() and also let $O_j \in C_N^i$, i.e., a normal group for a given relevant attribute.
Now, as per definition, the attack group, i.e., C_A given by SD_attrib_clus() is the intersection of all those attribute clusters C_A^i which,

- *have high relevance for attack class over selected subset of relevant features and*
- *have high compactness.*

So if $O_j \in C_N^i$, none of the above two conditions are fulfilled. Hence the proof. □

Table 2: Symbol Table

Symbols Used	Their meaning
D	Dataset.
D_{a_i}	i^{th} Attribute of dataset D (i = 1, 2, ... , n).
C_k^i	Attribute cluster of i^{th} attribute (k=1,2).
CP_k^i	Compactness value for cluster C_k^i
S	Subset of attributes.
O_j^i	j^{th} object of i^{th} attribute (j=1,2,...,m).
n	Total number of attributes
m	Total number of objects
C_A^i	Attack cluster for i^{th} attribute.
C_N^i	Normal cluster for i^{th} attribute.
C_A	Final attack cluster

3.1 Proposed Framework

The proposed framework for the detection of XSS attack is shown in Figure 3. We have proposed a proxy based approach, where it is attempted to balance the load in both the client and the server. A majority of the detection task is carried out in the proxy. An initial check for vulnerability is done in the client side.

A. **Client-based Processing:** An initial checking for the vulnerability is carried out at the client machine. Though one of our objects is to balance the work load between the client and the server, considering the possible low computational ability of a client, we maintain minimum overhead in the client machine. We assign three tasks to the client, i.e., preprocessing, feature extraction of the captured data and $\alpha-$divergence test. The client machine also maintains the profiles of attack and normal instances provided by the detection module in the proxy for reference. When the client sends a request to the server, it is handled by the client for preprocessing, feature extraction and $\alpha-$divergence test with reference to the attack/normal profile. If the value exceeds a predefined threshold value then the request is not further processed. It is dropped in the client side only. Otherwise, the request is forwarded to the proxy for further processing.

B. **Proxy-level Processing:** The majority of the detection tasks are carried out in the proxy server to keep the load in the main server minimum. This includes a step-by-step method to detect the attack using an unsupervised approach. The method follows four steps in sequence, viz., (a) data gathering, (b) preprocessing and feature extraction, (c) feature selection using an ensemble approach and (d) attack detection using attribute clustering over an optimal subset of relevant features. The steps are discussed in detail next.

B.1 ***Data Gathering:*** A major brainstorming task of this proxy level processing is to find the Websites for gathering the attack scripts. Since most of the Websites remove the scripts as they are no longer in use once detected, so finding such scripts are difficult. We have collected most of the attack scripts from [6]. Similarly, we gather the normal scripts using a testbed in our institution.

B.2 ***Preprocessing and Feature Extraction:*** This step involves finding out a number of features to describe the gathered data. We have found a total of 15 features relevant to our problem as also can be found in [22]. After extracting the features a 16 dimensional dataset (including the class label) is prepared. But since all the features in the dataset are not equally weighted and the ranges vary by a large margin, we have normalized the dataset using min-max normalization method.

B.3 ***Feature Selection Using Rank Aggregation:*** At this step, the features which are least relevant are excluded and only a subset of optimal relevant features is taken. Rank aggregation algorithm available in R package selects an optimal subset of attributes (say S) from total number of attributes n. Rank aggregation framework consists of a number of steps as shown in Figure 4. The prerequisite for the algorithm is a dataset with class labels which is given to different ranking based feature selection algorithms such as infogain [26], correlation based feature selection [14], gain Ratio [25], symmetric uncertainty [13], chi-square [18], mutual information [16] and reliefF [30]. The rankings given by these algorithm are input to a rank aggregation algorithm for the final subset of relevant features generation as shown in the Figure 5.

B.4 ***Attribute Clustering:*** Our proposed attribute clustering clusters the instances using the algorithm shown in Algorithm 1. Attribute clustering algorithm is based on the *kmeans* [15] clustering algorithm. Here each feature is clustered individually applying *kmeans*. The *kmeans* algorithm refers the parameters,

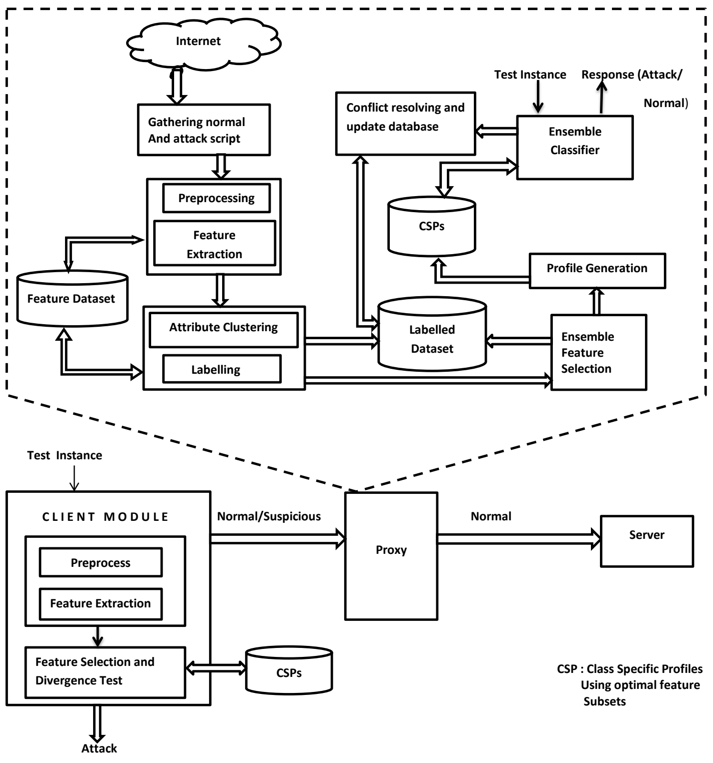


Figure 3: Proposed framework for XSS attack detection

viz., *indexMatrix* and sumd (as shown in Algorithm 1). The indexMatrix holds the cluster ids for the objects of an attribute. After that, cluster intersection is performed on the basis of the cluster compactness i.e., the more compact cluster of each attribute is considered. The attributes are taken on the basis of their rank given by the rank aggregation algorithm. After clustering, the groups are labeled using supervised approach w.r.t the already built profiles.

3.2 Algorithm for Attribute Clustering

The steps of the proposed attribute clustering algorithm is given in Algorithm 1.

3.3 Complexity Analysis

The complexity of the *SD_attrib_clus()* algorithm is primarily dominated by the *kmeans* clustering algorithm. All other operations are simple merging and intersection operations. So they are of $O(n \times m)$. Where $(n \times m)$ is the dimension of the original dataset. As we know, the complexity of *kmeans* algorithm is $O(n \times m^{(dk+1)} logm)$. Where $m \times n$ is the dimension of the dataset , d= dimension of the dataset given as input to the *kmeans* algorithm, and k=number of clusters. For our algorithm k=2 and d=1. Hence the complexity of our algorithm is $O(n \times m^3 logm)$.

4 Experimental Results

The experiments were carried out in both Windows 7 and Linux environment. The machine used was a 64-bit machine with 2 GB RAM. Matlab 2010 was used to performing attribute clustering. WEKA 3.7.11 was used to run the individual ranking algorithm on the labeled dataset. R package was used to run the rank aggregation algorithm over the rank lists given by the individual rankers. All the experiments carried out can be subdivided into the following sections.

4.1 Dataset Preparation

Dataset preparation involves several steps as described below.

4.1.1 Data Gathering

The first step of dataset generation is the collection of data from the Internet. Since the malicious scripts are immediately removed after detection from the Web applications, so it is very hard to collect live scripts. We have collected attack scripts from [6] and the benign JavaScripts from various Websites, which are using rich JavaScript contents. Figure 6 and Figure 7 are the example of collected attack and normal script respectively.

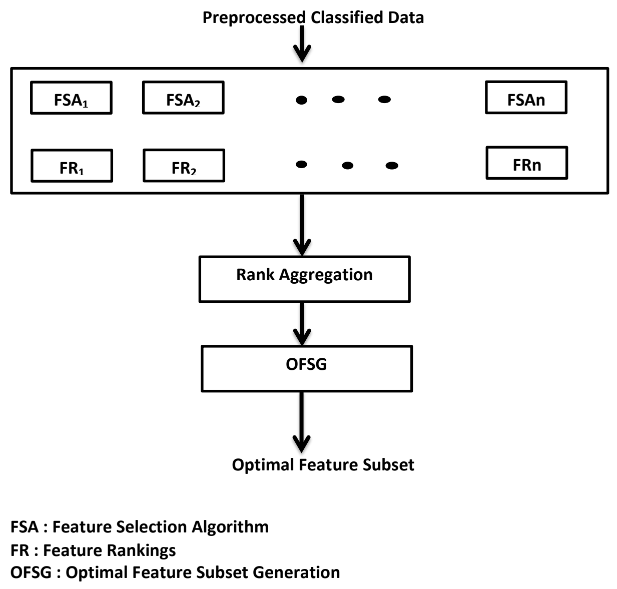

Preprocessed Classified Data
FSA₁
FSA₂
FSAn
FR₁
FR₂
FRn
Rank Aggregation
OFSG
Optimal Feature Subset
FSA : Feature Selection Algorithm
FR : Feature Rankings
OFSG : Optimal Feature Subset Generation

Figure 4: Optimal feature selection framework

4.1.2 Feature Extraction

After gathering the data, the second major phase is to extract the features that are relevant to our problem. After performing a thorough study of existing works [22] and the gathered data we finally picked up fifteen features as described in Table 3.

4.1.3 Modules

After extraction of the fifteen features a total of seventeen dimensional dataset (including the Sl. no. and class label) is prepared. However, while attribute clustering is performed only the first 15 features are considered as shown in Figure 8. The values of the instances for the 16^{th} feature in most cases are found to be zero. The dataset consists of 71 instances as of now and is flexible. That is, at any point of time if we find a new attack script or normal script we can add that instance to the existing dataset.

Different procedures written in C and their functions are described bellow.

1) *extract_script():* This function extract only the codes included within the script begin tag $< script >$ and the script end tag $< /script >$. All the codes other then this are discarded as they are not executed as JavaScript. The function also calls all the remaining methods.

2) *compute_length():* Calculates the total number of characters in the script.

3) *no_of_lines():* Calculates the total number of lines in the script.

4) *no_of_strings():* This function outputs the total number of lines in the script.

5) *avg_characters():* It calculates average number of characters per line in the script.

6) *percentage_whitespace():* This method gives the percentage of whitespace characters with respect to the total number of characters in the script.

7) *avg_string_length():* It calculates average length of the strings in terms of number of characters present in the string.

8) *no_of_comments():* This method gives the number of comment lines present in the script.

9) *avg_comments_per_line():* It calculates the average number of comments per line of the script.

10) *no_of_words():* Calculates the total number of words in the script.

11) *percentage_of_not_commented_words():* This method calculates the percentage of words that are not commented over the total number of words present in the script.

12) *count_hex_octal():* It outputs the total number of hexadecimal numbers and octal numbers present in the script.

13) *human_readability():* It gives the output in boolian form i.e., either 'Y'(Yes) or 'N'(No). If a script is human readable then it gives the output as 'Y', otherwise 'N'. Human readability is determined with the help of the following methods.

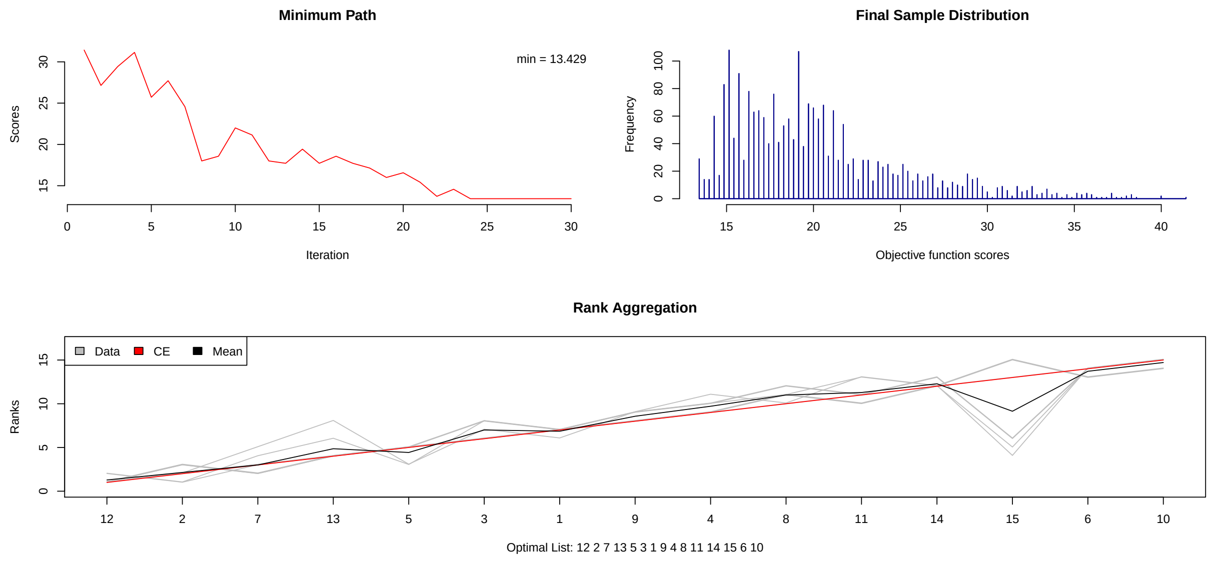


Figure 5: Output of the rank aggregation algorithm with optimal feature subset

a. *cal_percent_alphabetes():* Calculates the percentage of words where percentage of alphabets is >70%.

b. *cal_percent_vowels():* Calculates percentage of words where percentage of vowels lies in the range 20%-60%.

c. *percent_length():* Calculates the percentage of words which are less than 15 characters long.

d. *percent_repetition():* Calculates the percentage of words containing repetition of the same letter less than 3 times.

14) *methods_called():* This function gives the total number of methods called in the script.

15) *avg_arg_length():* Calculates the average argument length to each method.

16) *count_unicode_char():* Calculates the number of unicode characters present in the script.

4.1.4 Increase in the Population Density of the Dataset

After collecting the attack and normal scripts, with the help of the modules described in the previous subsection we created a dataset consisting of 71 attack and normal instances in the ratio 1:2 respectively. Now with the help of a module written in C, we increase the number of instances of the dataset to 1078 instances with the same ratio 1:2, respectively. This dataset is used to perform all the operations performed in the following sections.

4.1.5 Normalization of the Dataset

In many pragmatic scenarios, a dataset may consist of attributes or features having values with different ranges. It generally tends to create problem while the some of the values of some attributes are relatively much larger then that of the other attributes. This is because, larger values have a greater impact on the proximity measures like Euclidean distance. Since the base of our proposed attribute clustering algorithm is kmeans, which uses Euclidean distance measure, so it is very important for us to normalize the dataset. Figure 8 displays a part of the original dataset, whereas Figure 9 shows a part of the dataset after normalization. We have used min-max normalization to normalize the dataset. The formula for which is given next.

$$X_n = \frac{(X - X_{min})}{(X_{max} - X_{min})}.$$

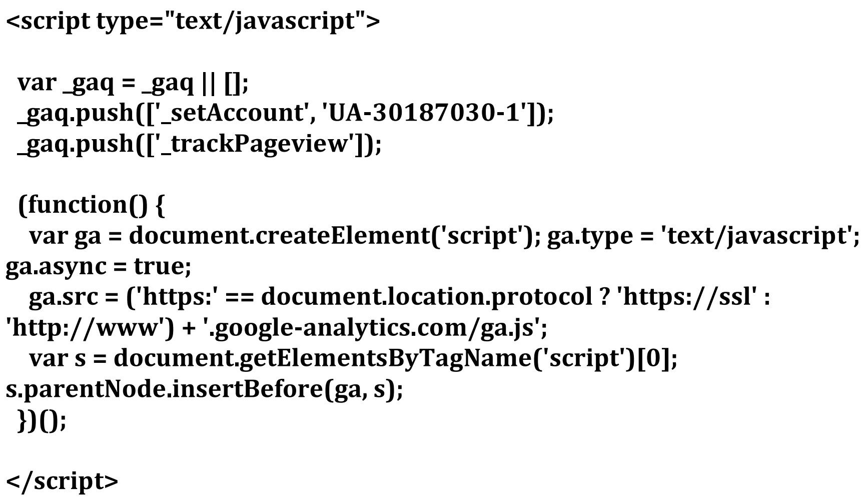
Where, X_n = Normalized value between 0 and 1, X = Original value, X_{max} = Maximum value of the attribute, X_{min} = Minimum value of the attribute.

4.2 Results

In this section, we have shown the true positive rate, false positive rate, and accuracy in identifying the groups of attack and normal scripts. An ROC curve is plotted as shown in Figure 10 to demonstrate the detection performance.

4.2.1 ROC Curve

Receiver Operating Curve (ROC) for our dataset is the curve of True Positive Rate (TPR) vs False Positive rate (FPR) of the clusters given by different subset of the attributes or features. Table 4 shows the value of the TPR, FPR, and accuracy of the clusters given by the attribute selection algorithm based on the feature subsets. The feature subsets contains the feature values according to the rank given by the ensemble feature selection algorithm.



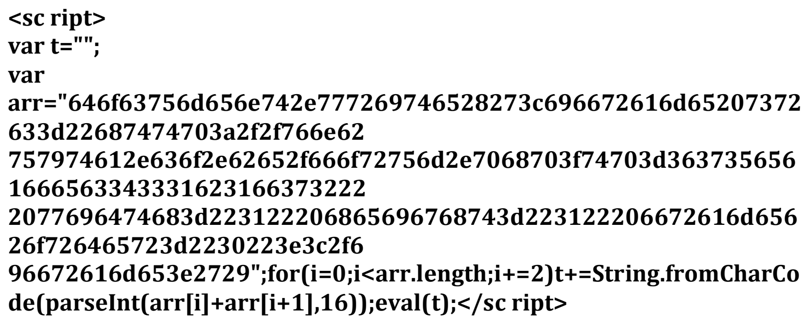
```
<script type="text/javascript">

 var _gaq = _gaq || [];
 _gaq.push(['_setAccount', 'UA-30187030-1']);
 _gaq.push(['_trackPageview']);

 (function() {
   var ga = document.createElement('script'); ga.type = 'text/javascript';
ga.async = true;
   ga.src = ('https:' == document.location.protocol ? 'https://ssl' :
'http://www') + '.google-analytics.com/ga.js';
   var s = document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(ga, s);
 })();

</script>
```

Figure 6: Example of a benign JavaScript



```
<sc ript>
var t="";
var
arr="646f63756d656e742e777269746528273c696672616d65207372
633d22687474703a2f2f766e62
757974612e636f2e62652f666f72756d2e7068703f74703d363735656
16665633433316231663733222
2077696474683d223122206865696768743d223122206672616d656
26f726465723d2230223e3c2f6
96672616d653e2729";for(i=0;i<arr.length;i+=2)t+=String.fromCharCo
de(parseInt(arr[i]+arr[i+1],16));eval(t);</sc ript>
```

Figure 7: Example of an attack script

Steps involved in calculating the True Positive (TP) and False Positive (FP) values of a cluster given by the attribute clustering algorithm are as follows:

- The attribute rank subset given by rank aggregation is taken and SD_attrib_clus() algorithm is applied on the whole dataset according to the given feature rank.
- The cluster which is more compact is considered as the attack cluster and the cluster instances are stored in a matrix.
- Now from the actual labeled dataset the attack instances are determined and intersection of these instances with the previously stored cluster instances are found. Thus we get the TP value. And the instances, that are excluded are counted as the FP value.
- The TPR and FPR are calculated from these TP and FP values with the help of the following formulas.

$$\text{True Positive Rate(TPR)} = \frac{\sum True\ Positive}{\sum Condition\ Positive}$$

$$\text{False Positive Rate(FPR)} = \frac{\sum False\ Positive}{\sum Condition\ Negative}$$

$$\text{Accuracy(ACC)} = \frac{\sum True\ Positive + \sum True\ Negative}{\sum Total\ Instances}$$

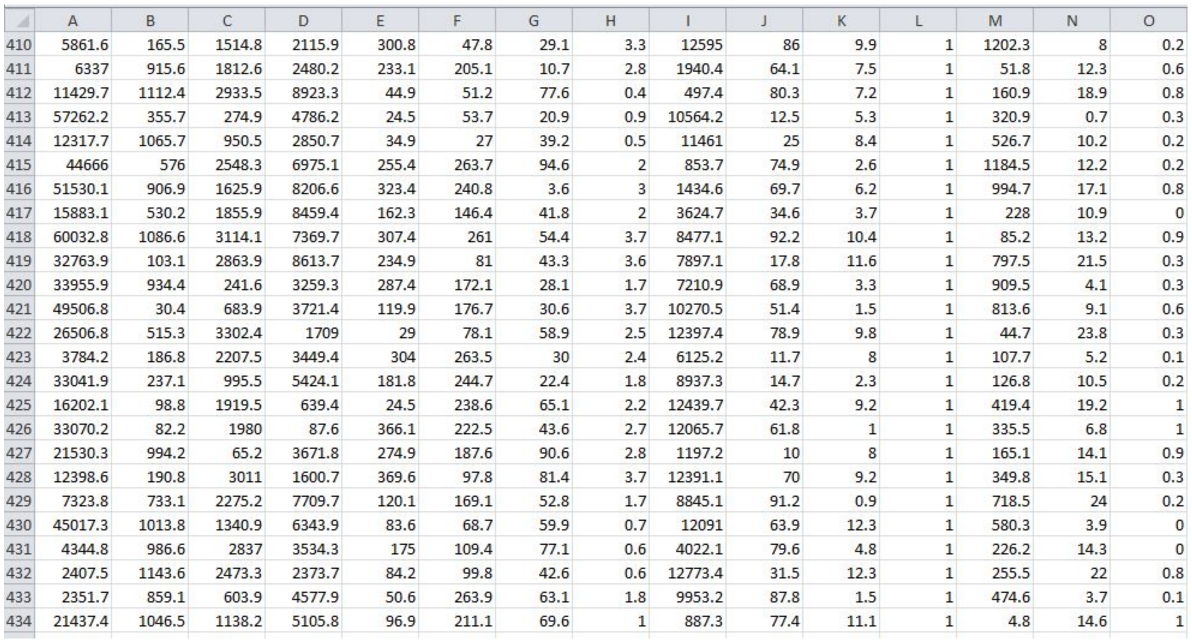
4.3 Comparison with Other Methods

In this section, we compare our method with other competing methods of XSS detection.

- Like [22], our method is also established on feature dataset generated based on the extracted features from attack and normal scripts.
- Like [7, 29], we also evaluate our method in terms of detection accuracy and the performance of our method is highly satisfactory.
- Unlike [7, 22], our method uses unsupervised attribute clustering technique to group the JavaScripts into legitimate and malicious.
- Unlike most other methods [21, 33], our approach attempts to balance the load between the client and server.

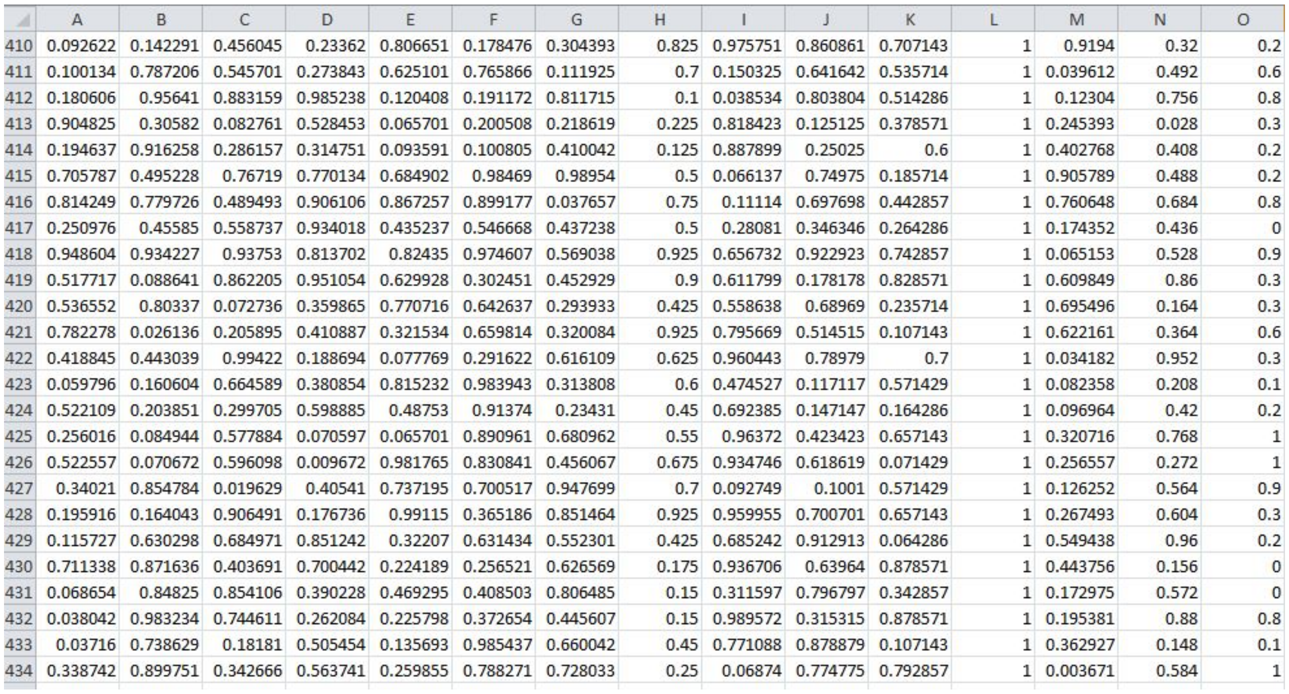
Table 3: Description of extracted features

Sl No.	Feature Label	Feature Description
1	A	Number of characters in the script
2	B	Number of lines in the Script
3	C	Number of strings in the script
4	D	Average characters per line
5	E	Percentage of whitespace in the script
6	F	Average string length
7	G	Number of comments in the script
8	H	Average comments per line
9	I	Total number of words
10	J	Percentage of words that are not commented
11	K	Number of octal numbers
12	L	Human readability in terms of yes or no. checking criteria are:
		a)Percentage of words which are >70% alphabetical >=45%
		b)Percentage of words, where 20% < vowels<60% >=40%
		c)Percentage of words which are less than 15 characters long>=70%
		d)Percentage of words containing< 3 repetition of the same letter in a row>=80%
13	M	Number of methods called
14	N	Average argument length
15	O	Number of unicode symbols
16	P	Number of HEX numbers



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
410	5861.6	165.5	1514.8	2115.9	300.8	47.8	29.1	3.3	12595	86	9.9	1	1202.3	8	0.2
411	6337	915.6	1812.6	2480.2	233.1	205.1	10.7	2.8	1940.4	64.1	7.5	1	51.8	12.3	0.6
412	11429.7	1112.4	2933.5	8923.3	44.9	51.2	77.6	0.4	497.4	80.3	7.2	1	160.9	18.9	0.8
413	57262.2	355.7	274.9	4786.2	24.5	53.7	20.9	0.9	10564.2	12.5	5.3	1	320.9	0.7	0.3
414	12317.7	1065.7	950.5	2850.7	34.9	27	39.2	0.5	11461	25	8.4	1	526.7	10.2	0.2
415	44666	576	2548.3	6975.1	255.4	263.7	94.6	2	853.7	74.9	2.6	1	1184.5	12.2	0.2
416	51530.1	906.9	1625.9	8206.6	323.4	240.8	3.6	3	1434.6	69.7	6.2	1	994.7	17.1	0.8
417	15883.1	530.2	1855.9	8459.4	162.3	146.4	41.8	2	3624.7	34.6	3.7	1	228	10.9	0
418	60032.8	1086.6	3114.1	7369.7	307.4	261	54.4	3.7	8477.1	92.2	10.4	1	85.2	13.2	0.9
419	32763.9	103.1	2863.9	8613.7	234.9	81	43.3	3.6	7897.1	17.8	11.6	1	797.5	21.5	0.3
420	33955.9	934.4	241.6	3259.3	287.4	172.1	28.1	1.7	7210.9	68.9	3.3	1	909.5	4.1	0.3
421	49506.8	30.4	683.9	3721.4	119.9	176.7	30.6	3.7	10270.5	51.4	1.5	1	813.6	9.1	0.6
422	26506.8	515.3	3302.4	1709	29	78.1	58.9	2.5	12397.4	78.9	9.8	1	44.7	23.8	0.3
423	3784.2	186.8	2207.5	3449.4	304	263.5	30	2.4	6125.2	11.7	8	1	107.7	5.2	0.1
424	33041.9	237.1	995.5	5424.1	181.8	244.7	22.4	1.8	8937.3	14.7	2.3	1	126.8	10.5	0.2
425	16202.1	98.8	1919.5	639.4	24.5	238.6	65.1	2.2	12439.7	42.3	9.2	1	419.4	19.2	1
426	33070.2	82.2	1980	87.6	366.1	222.5	43.6	2.7	12065.7	61.8	1	1	335.5	6.8	1
427	21530.3	994.2	65.2	3671.8	274.9	187.6	90.6	2.8	1197.2	10	8	1	165.1	14.1	0.9
428	12398.6	190.8	3011	1600.7	369.6	97.8	81.4	3.7	12391.1	70	9.2	1	349.8	15.1	0.3
429	7323.8	733.1	2275.2	7709.7	120.1	169.1	52.8	1.7	8845.1	91.2	0.9	1	718.5	24	0.2
430	45017.3	1013.8	1340.9	6343.9	83.6	68.7	59.9	0.7	12091	63.9	12.3	1	580.3	3.9	0
431	4344.8	986.6	2837	3534.3	175	109.4	77.1	0.6	4022.1	79.6	4.8	1	226.2	14.3	0
432	2407.5	1143.6	2473.3	2373.7	84.2	99.8	42.6	0.6	12773.4	31.5	12.3	1	255.5	22	0.8
433	2351.7	859.1	603.9	4577.9	50.6	263.9	63.1	1.8	9953.2	87.8	1.5	1	474.6	3.7	0.1
434	21437.4	1046.5	1138.2	5105.8	96.9	211.1	69.6	1	887.3	77.4	11.1	1	4.8	14.6	1

Figure 8: A portion of the original dataset



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
410	0.092622	0.142291	0.456045	0.23362	0.806651	0.178476	0.304393	0.825	0.975751	0.860861	0.707143	1	0.9194	0.32	0.2
411	0.100134	0.787206	0.545701	0.273843	0.625101	0.765866	0.111925	0.7	0.150325	0.641642	0.535714	1	0.039612	0.492	0.6
412	0.180606	0.95641	0.883159	0.985238	0.120408	0.191172	0.811715	0.1	0.038534	0.803804	0.514286	1	0.12304	0.756	0.8
413	0.904825	0.30582	0.082761	0.528453	0.065701	0.200508	0.218619	0.225	0.818423	0.125125	0.378571	1	0.245393	0.028	0.3
414	0.194637	0.916258	0.286157	0.314751	0.093591	0.100805	0.410042	0.125	0.887899	0.25025	0.6	1	0.402768	0.408	0.2
415	0.705787	0.495228	0.76719	0.770134	0.684902	0.98469	0.98954	0.5	0.066137	0.74975	0.185714	1	0.905789	0.488	0.2
416	0.814249	0.779726	0.489493	0.906106	0.867257	0.899177	0.037657	0.75	0.11114	0.697698	0.442857	1	0.760648	0.684	0.8
417	0.250976	0.45585	0.558737	0.934018	0.435237	0.546668	0.437238	0.5	0.28081	0.346346	0.264286	1	0.174352	0.436	0
418	0.948604	0.934227	0.93753	0.813702	0.82435	0.974607	0.569038	0.925	0.656732	0.922923	0.742857	1	0.065153	0.528	0.9
419	0.517717	0.088641	0.862205	0.951054	0.629928	0.302451	0.452929	0.9	0.611799	0.178178	0.828571	1	0.609849	0.86	0.3
420	0.536552	0.80337	0.072736	0.359865	0.770716	0.642637	0.293933	0.425	0.558638	0.68969	0.235714	1	0.695496	0.164	0.3
421	0.782278	0.026136	0.205895	0.410887	0.321534	0.659814	0.320084	0.925	0.795669	0.514515	0.107143	1	0.622161	0.364	0.6
422	0.418845	0.443039	0.99422	0.188694	0.077769	0.291622	0.616109	0.625	0.960443	0.78979	0.7	1	0.034182	0.952	0.3
423	0.059796	0.160604	0.664589	0.380854	0.815232	0.983943	0.313808	0.6	0.474527	0.117117	0.571429	1	0.082358	0.208	0.1
424	0.522109	0.203851	0.299705	0.598885	0.48753	0.91374	0.23431	0.45	0.692385	0.147147	0.164286	1	0.096964	0.42	0.2
425	0.256016	0.084944	0.577884	0.070597	0.065701	0.890961	0.680962	0.55	0.96372	0.423423	0.657143	1	0.320716	0.768	1
426	0.522557	0.070672	0.596098	0.009672	0.981765	0.830841	0.456067	0.675	0.934746	0.618619	0.071429	1	0.256557	0.272	1
427	0.34021	0.854784	0.019629	0.40541	0.737195	0.700517	0.947699	0.7	0.092749	0.1001	0.571429	1	0.126252	0.564	0.9
428	0.195916	0.164043	0.906491	0.176736	0.99115	0.365186	0.851464	0.925	0.959955	0.700701	0.657143	1	0.267493	0.604	0.3
429	0.115727	0.630298	0.684971	0.851242	0.32207	0.631434	0.552301	0.425	0.685242	0.912913	0.064286	1	0.549438	0.96	0.2
430	0.711338	0.871636	0.403691	0.700442	0.224189	0.256521	0.626569	0.175	0.936706	0.63964	0.878571	1	0.443756	0.156	0
431	0.068654	0.84825	0.854106	0.390228	0.469295	0.408503	0.806485	0.15	0.311597	0.796797	0.342857	1	0.172975	0.572	0
432	0.038042	0.983234	0.744611	0.262084	0.225798	0.372654	0.445607	0.15	0.989572	0.315315	0.878571	1	0.195381	0.88	0.8
433	0.03716	0.738629	0.18181	0.505454	0.135693	0.985437	0.660042	0.45	0.771088	0.878879	0.107143	1	0.362927	0.148	0.1
434	0.338742	0.899751	0.342666	0.563741	0.259855	0.788271	0.728033	0.25	0.06874	0.774775	0.792857	1	0.003671	0.584	1

Figure 9: A portion of the normalized dataset

Table 4: Accuracy of the classes based on the feature subset

Feature Subset	True Positive Rate(TPR)	False Positive Rate(FPR)	Accuracy
12,2,7,13,5,3,1,9,4,8,11,14,15,6,10	0.24	0	0.7449
12,2,7,13,5,3,1,9,4,8,11,14,15,6	0.517	0.003	0.8358
12,2,7,13,5,3,1,9,4,8,11,14,15	0.978	0.006	0.9889
12,2,7,13,5,3,1,9,4,8,11,14	0.983	0.006	0.9907
12,2,7,13,5,3,1,9,4,8,11	0.983	0.006	0.9907
12,2,7,13,5,3,1,9,4,8	0.983	0.006	0.9907
12,2,7,13,5,3,1,9,4	0.983	0.007	0.9897
12,2,7,13,5,3,1,9	0.983	0.007	0.9897
12,2,7,13,5,3,1	0.983	0.007	0.9897
12,2,7,13,5,3	0.983	0.007	0.9897
12,2,7,13,5	0.983	0.007	0.9897
12,2,7,13	0.983	0.007	0.9897
12,2,7	0.983	0.008	0.9889
12,2	0.983	0.008	0.9889

```
Data: D = Dataset, $D_{a_i} = i^{th}$ attribute of D,
      $\forall i = 1, 2, ..., n$
K = No. of clusters
Result: $C_A$ = Attack cluster
Function SD_attrib_clus()
foreach attribute $D_{a_i} \in D$ do
    [indexMatrix, sumd] = kmeans($D_{a_i}$, K)
    foreach Cluster $C^i_k, k = 1, 2$ do
        $CP^i_k = \frac{sumd}{No.\,of\,objects\,in\,C^i_k}$
    end
    if $CP^i_k < CP^i_{k+1}$, k=1 then
        foreach Object $O^i_j \in D_{a_i}$ do
            if indexMatrix($O^i_j$) == k then
                $C^i_A \leftarrow O^i_j$
            end
            else
                $C^i_N \leftarrow O^i_j$
            end
        end
    end
    else
        foreach Object $O^i_j \in D_{a_i}$ do
            if indexMatrix($O^i_j$) == k+1 then
                $C^i_A \leftarrow O^i_j$
            end
            else
                $C^i_N \leftarrow O^i_j$
            end
        end
    end
end
// Find the common objects of the attributes
   in the order given by rank aggregation
   method
$C_A = \cap C^i_A, \forall i = 1, 2, ..., S, S \leq n$
```

Algorithm 1: Attribute clustering algorithm

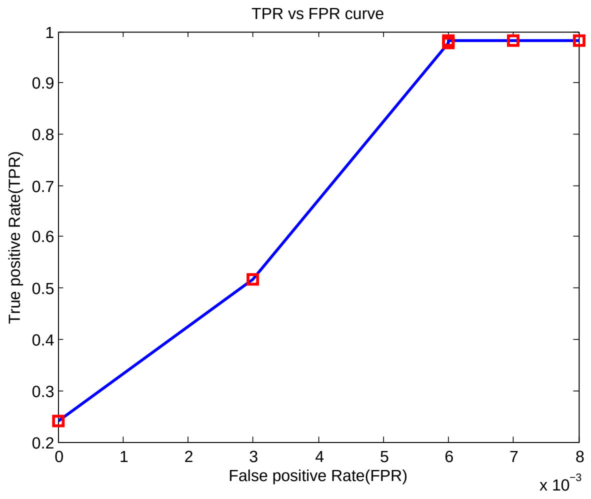


Figure 10: ROC curve

5 Discussion and Conclusion

Based on the results we got from the attribute clustering algorithm, proceeded by rank aggregation using cross entropy monte carlo algorithm, shows us a way how we can use unsupervised techniques in clustering the malicious and benign scripts into two classes with high accuracy. The computation overhead also decreases significantly in the proxy as our proposed method distributes the task between the client and the server. The detection mechanism in the proxy is easy to implement and requires a little knowledge to detect an attack with high accuracy.

Acknowledgments

The authors would like to thank the Ministry of HRD, Govt. of India for funding as a Centre of Excellence with thrust area in Machine Learning Research and Big Data Analytics for the period 2014-2019.

References

[1] E. Adi, "A design of a proxy inspired from human immune system to detect SQL injection and cross-site scripting," *Procedia Engineering*, vol. 50, pp. 19–28, 2012.

[2] E. Athanasopoulos, A. Krithinakis, and E. P. Markatos, "Hunting cross-site scripting attacks in the network," in *Third International Conference on Advanced Computing (ICoAC'11)*, pp. 89–92, 2011.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy (SP'08)*, pp. 387–401, 2008.

[4] D. K. Bhattacharyya and J. K. Kalita, *Network anomaly detection: A machine learning perspective*, CRC Press, 2013.

[5] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Survey on incremental approaches for network anomaly detection," *International Journal of Communication Networks and Information Security*, vol. 3, no. 3, pp. 226–239, 2011.

[6] Blwood, *Multiple XSS Vulnerabilities in Tikiwiki 1.9.x*, 2006. (`http://www.securityfocus.com/archive//435127`)

[7] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler of a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International Conference on World Wide Web*, pp. 197–206, 2011.

[8] S. Christey, *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, 2011. (`http://cwe.mitre.org/top25`)

[9] S. Chun, C. Jing, H. ChangZhen, X. JingFeng, W. Hao, and M. Raphael, "A xss attack detection method based on skip list," *International Journal of Security and Its Applications*, vol. 10, no. 5, pp. 95–106, 2008.

[10] J. Grossman, *XSS Attacks: Cross-site scripting exploits and defense*, Syngress, 2007.

[11] B. B. Gupta, S. Gupta, S. Gangwar, M. Kumar, and P. K. Meena, "Cross-site scripting (XSS) abuse and defense: exploitation on several testing bed environments and its defense," *Journal of Information Privacy and Security*, vol. 11, no. 2, pp. 118–136, 2015.

[12] S. Gupta and B. B. Gupta, "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, no. 3, pp. 897–920, 2016.

[13] M. A. Hall, *Correlation-based Feature Selection for Machine Learning*, Doctoral Dissertation, The University of Waikato, 1999.

[14] M. A. Hall and L. A. Smith, "Feature selection for machine learning: Comparing a correlation-based filter approach to the wrapper," in *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference*, pp. 235–239, 1999.

[15] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[16] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "MIFS-ND: a mutual information-based feature selection method," *Expert Systems with Applications*, vol. 41, no. 14, pp. 6371–6385, 2014.

[17] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in ddos attacks: trends and challenges," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2242–2270, 2015.

[18] L. Huan and R. Setiono, "Chi2: feature selection and discretization of numeric attributes," in *Proceedings of Seventh International Conference on Tools with Artificial Intelligence*, pp. 388–391, 1995.

[19] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proceedings of the 16th International Conference on World Wide Web*, pp. 601–610, New York, NY, USA, 2007.

[20] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *IEEE Symposium on Security and Privacy*, pp. 6–263, 2006.

[21] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, pp. 330–337, New York, NY, USA, 2006.

[22] P. Likarish, J. Eunjin, and J. Insoon, "Obfuscated malicious javascript detection using classification techniques," in *4th International Conference on Malicious and Unwanted Software (MALWARE'09)*, pp. 47–54, 2009.

[23] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *26th Annual International Telecommunications Energy Conference*, pp. 71–80, 2004.

[24] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th International Conference on World Wide Web*, pp. 432–441, New York, NY, USA, 2005.

[25] J. R. Quinlan, *C4. 5: Programs for Machine Learning*, Elsevier, 2014.

[26] D. Roobaert, G. Karakoulas, and N. V. Chawla, "Information gain, correlation and support vector machines," in *Feature Extraction*, pp. 463–470, 2006.

[27] S. Saha, "Consideration points detecting cross-site scripting," *International Journal of Computer Science and Information Security*, vol. 4, no. 1 & 2, Aug. 2009.

[28] M. I. P. Salas and E. Martins, "Security testing methodology for vulnerabilities detection of XSS in web services and ws-security," *Electron Notes in Theoritical Computer Science*, vol. 302, pp. 133–154, 2014.

[29] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis," in *Proceedings of International Conference on Software Engineering*, pp. 642–651, Piscataway, NJ, USA, 2013.

[30] Y. Wang and F. Makedon, "Application of relief-f feature filtering algorithm to selecting informative genes for cancer classification using microarray data," in *IEEE Computational Systems Bioinformatics Conference*, pp. 497–498, 2004.

[31] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceeding of ACM/IEEE 30th International Conference on Software Engineering*, pp. 171–180, 2008.

[32] D. Wichers, *OWASP, The Open Web Application Security Project*, 2013. (`http://www.owasp.org`)

[33] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "SWAP: Mitigating XSS attacks using a reverse proxy," in *Proceeding of 5th International Workshop on Software Engineering for Secure Systems*, IEEE Computer Society, 2009.

Biography

Swaswati Goswami obtained Master of Technology degree in Information Technology from Tezpur University, India in the year 2012. Her research interests are machine learning and network security.

Nazrul Hoque obtained Master of Technology degree in Information Technology from Tezpur University, India in the year 2012. Currently, he is a PhD candidate in the Department of Computer Science and Engineering at Tezpur University. His research interests are machine learning and network security.

Dhruba K Bhattacharyya received his Ph.D. in Computer Science from Tezpur University in 1999. He is a Professor in the Computer Science & Engineering Department at Tezpur University. His research areas include data mining, bioinformatics, network security, and big data analytics. Prof. Bhattacharyya has published 220+ research papers in the leading international journals and conference proceedings. In addition, Dr Bhattacharyya has written/edited 8 books. His book on Network Anomaly Detection: A Machine Learning Perspective is now popular among the network security researchers. Professor Bhattacharyya is Project Investigator of several prestigious major research grants such as Ministry of HRD's Center of Excellence under FAST, Center of High Performance Computing and UGC SAP DRS II of Govt. of India.