Design and Development of Anti-XSS Proxy

Hossain Shahriar\textsuperscript{1}, Sarah North\textsuperscript{1}, Wei-Chuen Chen\textsuperscript{2}, and Edward Mawangi\textsuperscript{2}

Department of Computer Science
Kennesaw State University
Kennesaw, Georgia 30144 USA
\textsuperscript{1}\{hshahria, snorth\}@kennesaw.edu
\textsuperscript{2}\{wchen10, emb3\}@students.kennesaw.edu

Abstract - Cross-Site Scripting (XSS) vulnerability allows an attacker to inject arbitrary JavaScript code that can be executed in the victim’s browser and cause security breaches. The discovery of XSS is still widespread among today’s web applications. As a result, there is a need to improve existing solutions or develop novel attack detection techniques. This paper proposes a proxy-level design and development of XSS attack detection approach (Anti-XSS) based on Kullback-Leibler Divergence (KLD) measure. The proposed approach has been applied for a number of open source PHP web applications containing XSS vulnerabilities. The initial results show that the approach can effectively detect XSS attacks and suffer from low false positive rate depending on the choice of threshold values of KLD.

Keywords: Kullback-Leibler Divergence, Information theory, Cross-Site Scripting, Web security.

I. INTRODUCTION

Cross-Site Scripting (XSS) has been ranked among top three vulnerabilities over the last few years \cite{1}. XSS vulnerability opens up the possibility for an attacker to inject arbitrary JavaScript code \cite{2} that can execute in the context of a victim’s browser. The injected script code cause unwanted behaviors (generating pop-up window) and security breaches (session hijacking \cite{3}).

Despite the presence of many mitigation approaches for XSS attacks at both client and server-sides \cite{4-15, 28-32}, the discovery of XSS vulnerability is still widespread. Thus, there is a need to improve existing solutions or develop novel attack detection techniques. In this paper, we design and develop a proxy-level XSS attack detection technique (Anti-XSS) based Kullback-Leibler Divergence (KLD) measure. Our intuition is that legitimate JavaScript code present in web applications should remain similar or very close to the JavaScript code of a rendered web page. A high deviation between the two sets may indicate XSS attacks.

We apply the proposed XSS attack detection approach for web applications implemented in PHP and containing XSS vulnerabilities. The initial results show that the approach can detect most of the known XSS attack signatures and show very low false positive warning subject to the choice of KLD divergence level between two sets of JavaScript. The proposed approach can handle different types of injected JavaScript code such as inline, URL attribute, and Cascading Style Sheet (CSS).

This paper is organized as follows. Section II shows an example of XSS attack followed by a brief introduction of related work. In Section III, the proposed KLD-based XSS attack detection proxy-level framework is discussed. Section IV describes the experimental results. Finally, Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

In this section, we first show an example of XSS attack in Section A followed by related work on XSS attack detection in Section B. We also discuss some related work from KLD from the literature in Section C.

A. An example of XSS attack

Figure 1 (a) shows the HTML code of a web page that accepts the user name in an HTML form. Here, a user can supply his/her name in a text box for displaying the name in PHP script. The supplied input is accessed in \texttt{show.php} shown in Figure 1(b). Note that \$\_POST["user"] variable retrieves the supplied input and then displays it in the response page (echo statement). The displayed content is not being filtered for possible JavaScript code. If a malicious user supplies arbitrary JavaScript code, it would execute when the response page is displayed in the victim’s browser.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{(A) HTML FORM} & \textbf{(B) SCRIPT (SHOW.PHP)} \\
\hline
\begin{verbatim}
<form action = "show.php" method="post">
<input type = "text" name = "user" value = "">
<input type = "submit" name = "submit" value = "Submit">
</form>
\end{verbatim} & \begin{verbatim}
Hello <\script>echo $\_POST["user"];</script>.
</verbatim} \\
\hline
\end{tabular}
\caption{HTML code and XSS vulnerable PHP code}
\end{table}

For example, if the user name is supplied as \texttt{alert (document.cookie)}</script>, then browser will display the cookie information. This is an example of reflected XSS attack. The other common variant is known as stored XSS attack where injected payloads are stored at the server side storage system. The payloads are retrieved at a later time when a victim sends requests from the client side.

B. Related work on XSS detection

XSS attack has been detected at both server and client sides. Shar et al. \cite{4} develop a testing approach by extracting and modeling the implemented defenses against XSS attacks as control flow graph, followed by checking the adequacy of defense mechanisms. In contrast, we proposed an approach at the proxy level to detect XSS attacks. Our approach can be
a complementary defense for applications having inadequate defense mechanisms against XSS attacks.

Frenz et al. [5] develop an IDS to capture a legitimate web page and extract all executable JavaScript code followed by generating a hash. At a later time, when the web page is generated, the extracted code is used to generate a hash and compare with the earlier generated hash value. A mismatch is used to flag an XSS attack at the IDS level. A similar approach has been introduced earlier in [6] which compares script code based on generated hash values. In contrast, our approach works at the proxy level and compare legitimate and observed JavaScript code based on KLD.

Gundy et al. [9] apply HTML element namespace randomization at the server side followed by derandomizing at the client side to prevent the injection of arbitrary JavaScript code. Similarly, Wurzinger et al. [10] encode all legitimate JavaScript function calls as syntactically invalid code so that attacker injected code gets executed. Nadji et al. [11] detect XSS attacks through the notion of document structure integrity validation where expected and actual parse trees of DOMs are compared. Johns et al. [12] detect XSS attacks by analyzing requests and response pages. In [28], the authors detect XSS attacks by comparing HTML code features based on server-side filtering of response pages.

Several works apply dynamic analysis at the server side (using tainted data flow-based analysis) to detect XSS attacks. In particular, library APIs that are invoked can be intercepted to track the presence of suspected inputs having XSS attack payload [13, 14, 15] to detect XSS attacks. The idea is to mark inputs as tainted, propagate taint information in program operations, and check specific program locations where tainted data is used. However, the analysis requires reimplementation of library APIs and server-side interpreters so that program operations become taint-aware.

A number of works propose XSS attack detection techniques at the client-side. Kirda et al. [7] develop a client-side proxy to prevent XSS attacks based on a set of rules applicable for a firewall. The rules allow or block incoming and outgoing webpage requests to prevent XSS attacks that may open new connections to designated attacker servers. However, the approach is not precise enough for detecting other variants of XSS including method overriding attacks and DOM-based XSS. Iha et al. [8] prevent XSS attacks by first rendering the tree structure (DOM view) of a web page followed by binding parameters at tree nodes so that the nodes are not interpreted. This approach requires the specific HTTP headers to be sent from the server side and be treated at the client-side.

XSS attacks can be detected by generating suitable test cases using mutation-based testing approach [29]. Interested readers can see [30] for a comprehensive discussion of XSS attack mitigation approaches. We are also aware of a number of open source scanner tools that can check XSS vulnerabilities [31, 32]. Our proposed approach can be considered as complementary to all the above mentioned approaches. Being our approach relying on KLD, the detection is anomaly-based. Hence, it brings the additional advantage of detecting unknown XSS attack signatures.

### C. Related work on KLD

Our work is motivated by a number of works that apply the concept of Kullback-Leibler Divergence (KLD) as a measure to solve a number of problems from various domains including document’s author identification [17], masquerade attack detection [21], and outlier data value detection in wireless sensor network [22]. Briggs [17] applies KLD to identify authorship of documents. They first build model of each author by aggregating documents generated by the author. Then, for a given document of unknown author, the approach finds the smallest KLD between a known model and the document. The model that is closest to the document is selected as the author. Similar to this work, we apply constant back-off smoothing technique to address the missing elements (or tokens derived from JavaScript code). Specifically, we compare the KLD between the JavaScript code from a generated response page and the expected script code from the page source. The deviation, if exceeds a given threshold value, provides an indication of the presence of injected JavaScript code in a response page.

Tapiador et al. [21] detect masquerade attacks based on an anomaly-based technique which compares a given request with known normal request using KLD measure. In a masquerade attack, an attacker steals credentials of legitimate users and performs further malicious actions using the credentials. The KLD enables the detection of padding in command sequences independent of the length and position in a block of request. In contrast, we apply KLD for detecting XSS attacks where JavaScript code can be injected any point in a response page.

Li et al. [22] apply differential KLD to detect anomalous data value in wireless sensor networks. The network is divided into clusters. In each cluster, the sensors remain physically close to each other and sense similar values. The outlier values are detected using KLD. Fukui et al. [23] measures the similarity of events based on KLD and applied it in the domain of fuel-cell. Sarkar et al. [24] apply information theoretic metrics including KLD to measure the quality of modularization of non-object orientated software.

### III. PROXY-LEVEL DETECTION OF XSS ATTACKS

In this section, we first briefly introduce the proxy-level design of Anti-XSS framework for XSS attack detection in Section IIIA. We then discuss the KLD measurement formula and followed by the computation algorithm based on constant back-off (absolute discounting) technique.

#### A. Proxy-level framework for XSS attack detection

Figure 2 shows a diagram of the proposed proxy-level attack detection approach. In particular, the XSS Detector module (denoted as the proxy) is capable of intercepting requests from the client-side and analyzing response pages sent by the server-side. The workflow of the diagram is as follows:

First, a server-side page is requested from the client (step 1) and the proxy forwards it to the server-side (step 2). The server-side sends the response page and the set of legitimate

---

**Proc. of the 8th IEEE International Conference for Internet Technology and Secured Transactions (ICITST-December 2013)**
JavaScript code $P$. In particular, the token elements extracted from the JavaScript code along with their occurrence probabilities are supplied. During step 3, the proxy analyzes the response page for XSS attack detection. In particular, it identifies $Q$ containing the set of token elements and their occurrence probabilities (by tokenizing the present JavaScript code in the actual response page) and then computes the KLD. If the KLD exceeds a threshold value, then the page is suspected to contain XSS attack inputs. Otherwise, the page is forwarded to the client-side (step 4).

Note that $P$ is pre-computed for each of the server-side pages. In particular, it requires identifying all JavaScript code from the server-side pages followed by tokenizing them (based on common separator such as $<$, $>$, ;) to identify unique elements and their occurrence probability. Our current approach includes all inline JavaScript code (e.g., `<script>...</script>`, URL attribute (e.g., `<img src= "javascript:...">`), and CSS (e.g., `<div style= "background:url('javascript:...')">`) that may contain JavaScript code. If the JavaScript code is encoded, then it is converted to plain text to generate token. If a server-side script does not contain any JavaScript, then we add a dummy symbol $sl$ with probability 1 in $P\{sl:1\}$. This is to avoid having an empty $P$ set, which makes it impractical for computing the distance between $P$ and any other distribution. We discuss the computation of KLD in Section III.B.

![Figure 2. Proxy-level Anti-XSS framework.](image)

### B. Kullback-Leibler Divergence

The Kullback-Leibler Distance (KLD) computes the distance between two given probability distributions. Let us assume that $P$ and $Q$ represent two probability distributions, where $P = \{p_1, ..., p_n\}$ and $Q = \{q_1, ..., q_n\}$. Then, the KLD is defined as follows [16]:

$$KLD(P, Q) = \sum_{i=1}^{n} p_i \log_2 \left( \frac{p_i}{q_i} \right) \quad (i)$$

The following two constraints (Equations (ii) and (iii)) are satisfied for Equation (i):

1. $\sum_{i=1}^{n} p_i = 1 \quad (ii)$
2. $\sum_{i=1}^{n} q_i = 1 \quad (iii)$

The KLD can be viewed as the additional message-length required when using a code based on the target distribution ($Q$) compared to using a code based on the true distribution ($P$). Note that KLD is not symmetric (i.e., $KLD(P, Q) \neq KLD(Q, P)$). Also, $KLD(P, Q) = 0$, iff $P = Q$.

We compute the KLD between two sets (or probability distributions) of JavaScript code. We tokenize the script code into unique elements and compute their occurrence probability to form $P$ and $Q$. Here, $P$ is the legitimate JavaScript code present in an application page, and $Q$ is the observed JavaScript code present in the response page. The term $p_i \log_2 (p_i/q_i)$ can be rewritten as subtraction of two terms: $p_i \log_2 p_i - p_i \log_2 q_i$. While we compute KLD ($P$, $Q$), if any of the occurrence probability of $p$ or $q$ is zero, then its logarithm value becomes infinite, hence it makes it difficult to compute KLD between $P$ and $Q$.

In the literature, smoothing techniques [17, 18, 19, 20] are applied to avoid having infinite distance between two probability distributions. Among a few well-known techniques (Dirichlet [19], Jelinek-Mercer [20], constant back-off [18]), we apply the constant back-off smoothing technique (absolute discounting) to address the infinite KLD value problem. Here, the elements that are missing in either $P$ or $Q$ sets are added with a very negligible amount of constant probability. At the same time, the present elements are subtracted with probability values in a way so that Equations (ii) and (iii) remain valid for the revised set of probability distributions (denoted as $P'$ and $Q'$).

#### Algorithm Smoothed_KLD($P$, $Q$, $e$)

**Input:** $P =$ Actual distribution, $Q =$ Observed distribution, $e =$ constant probability

**Output:** $d =$ KLD between the smoothed population sets

1. $P = \{p_i\}$, where $p_i \in P$.
2. $CP = |P|$
3. $Q = \{q_i\}$, where $q_i \in Q$.
4. $CQ = |Q|$
5. $SU = PUQ$
6. $CU = |SU|$
7. Smooth $P$ based on the following steps to generated $P'$:
   - $p'_i = p_i - pc$ if the $i^{th}$ element is in $P$
   - $p'_i = e$, otherwise
8. Compute $pc$ by solving $\sum_{i=1}^{n} p'_i = 1.0$
9. Similarly generate $Q'$, and identify $qc$ by solving $\sum_{i=1}^{n} q'_i = 1.0$
10. $d = \sum_{i=1}^{n} p'_i \log_2 \left( \frac{p'_i}{q'_i} \right)$
11. return $d$

![Figure 3. KLD computation using absolute discount smoothing](image)

Figure 3 shows the pseudo code of the smoothing technique that allows computing the KLD between $P$ and $Q$ for a given $e$ (constant probability that is a very low number assigned for unobserved elements in $P$ and $Q$). Here, $CP$ (line2) and $CQ$ (line4) are the cardinality of $P$ and $Q$ sets, respectively. $SU$ is the union of $P$ and $Q$ sets (line5). $CU$ is the cardinality of $SU$ (line 6). Line 7 generates a smoothed set of the actual distribution of JavaScript code element $P'$.
from original population \( P \), where \( P' \) contains elements that are present in \( P \) (each element’s probability is subtracted by \( pc \)) and also elements not observed in \( P \) but present in \( Q \) (i.e., all elements belong to the \( SU-SP \) set) where each element has the probability \( e \). The value of \( pc \) is solved in Step 8. It can be shown that \( pc = e^*SU-SP/[SP] \). Line 9 applies smoothing for the \( Q \) set and generates \( Q' \), and it can be shown that \( qc = e^*SU-SQ/[SQ] \). The KLD value is then computed for \( P' \) and \( Q' \) in step 10.

**Algorithm XSS Detection \( t, e, P, R \)**

Input: \( t \) = KLD threshold, \( e \) = constant probability for occurring unseen event, \( R \) = response page
Output: \( flag \) = false
1. Extract JavaScript code from \( R \) and generate population \( Q \)
2. if Smoothed_KLD \( (P, Q, e) > t \)
3. \( flag = true \)
4. return \( flag \).

Figure 4. Algorithm for KLD-based detection of XSS

Figure 4 shows the XSS attack detection algorithm based on the KLD for smoothed probability distributions. It accepts a threshold value \( t \) for comparing the distance to flag the observed JavaScript code as malicious or benign, and a constant probability value \( e \). Note that KLD requires the presence of a non-empty \( P \) set which we fill with at least one benign and symbolic element \( (s1) \) if there is no JavaScript code present in a web page. Similarly, if the response page does not contain any JavaScript code, then we also add the same symbolic element \( (s1) \). This approach allows obtaining the KLD between \( P \) and \( Q \) as 0, and avoiding false positive warnings for benign web pages.

IV. EXPERIMENTAL EVALUATION

We implement the proposed proxy-level XSS detection approach by extending the Fiddler tool [25]. Fiddler is a web application debugging tool that can intercept request and response pages. We implement the necessary code and filters to analyze response pages. The web applications are instrumented to build the \( P \) set.

We evaluate the proposed approach for three open source PHP applications available from sourceforge.net. These applications have been reported to contain XSS vulnerabilities. The applications are PHP-Address book (address manager), Serendipity (blog management), and PHP-fusions (content management system).

**TABLE I. CHARACTERISTICS OF PHP APPLICATIONS**

<table>
<thead>
<tr>
<th>Application</th>
<th># of files</th>
<th>echo</th>
<th>Inline</th>
<th>URL</th>
<th>CSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>312</td>
<td>567</td>
<td>7</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>Serendipity</td>
<td>1,344</td>
<td>1,619</td>
<td>56</td>
<td>43</td>
<td>126</td>
</tr>
<tr>
<td>PHP-fusions</td>
<td>718</td>
<td>4555</td>
<td>80</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

Table I shows the characteristics of these applications that include the number of PHP files present in these applications (column 2), the number of \( echo \) statements for generating dynamic output (column 3), and the three types of JavaScript code (inline, URL attribute, and CSS) (columns 4-6).

We deploy the web applications in an Apache web server with MySQL server as the backend database. We visit the web applications so that we can reach the forms to potentially supply modified pages and access the HTML forms. We randomly choose five types of malicious script code from OWASP [26]. Table II shows the description of each type of XSS attack along with example payload (\( a_0-a_5 \)). The attacks include injecting JavaScript through HTML attributes and invoking \( onload (a_0) \), \( onmouseover (a_1) \) onerror \( (a_2) \) method calls, encoded URI \( (a_3) \), and code encoding \( (a_4) \).

**TABLE II. XSS ATTACK TYPES**

<table>
<thead>
<tr>
<th>Description</th>
<th>Attack payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS in Attributes - onload ( (a_0) )</td>
<td>&lt;body onload=alert('test1')&gt;</td>
</tr>
<tr>
<td>XSS in Attributes - onmouseover ( (a_1) )</td>
<td>&lt;b onmouseover = alert('Wufff!')&gt;</td>
</tr>
<tr>
<td>XSS in Attributes- onerror ( (a_2) )</td>
<td>&lt;img src=’<a href="http://url.to.file.which.not.exist%E2%80%99">http://url.to.file.which.not.exist’</a> onerror = alert(document.cookie)&gt;</td>
</tr>
<tr>
<td>XSS via Encoded URI ( (a_3) )</td>
<td>&lt;IMG SRC=javascript:alert('test2')&gt;</td>
</tr>
<tr>
<td>XSS using code encoding ( (a_4) )</td>
<td>&lt;META HTTP-EQUIV=’refresh’ CONTENT=’0;url = data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVz’&gt;</td>
</tr>
</tbody>
</table>

To identify vulnerable input locations, we rely on the reported XSS vulnerabilities from OSVDB\(^1\) [27]. Table III shows the list of OSVDB ID from which we identify the responsible pages and inject arbitrary JavaScript code. While enabling the Anti-XSS proxy, we examine whether the injected code can be flagged as malicious by the proxy or not to measure the effectiveness of our approach.

Table IV shows the computed KLD during our evaluation for XSS attack types. The second column shows the vulnerability ID from OSVDB. Columns 3-7 show the KLD for attacks \( a_0-a_5 \). The lowest and highest KLD values for \( PHP \) Address book are 0.35 and 15.55, respectively. Similarly, we observe the lowest and highest KLD values for Serendipity \((0.02, 15.93)\) and PHP-fusions \((0.03, 15.92)\). The KLD depends on the present JavaScript code in the vulnerable script files. Note that the probability \( e \) for back-off smoothing is 0.0001 during the evaluation.

---

\(^1\) We considered vulnerability reported between the years 2007-2013.
The overall number of benign inputs, total attack (TPs), false negative (FN), and true negative (TN) are randomly generated benign strings into HTML form fields, respectively. The highest FN occurs when the threshold value (should be close to zero).

We evaluate the false positive (FP) rate by supplying randomly generated benign strings into HTML form fields, and then viewing the inputs by visiting the appropriate web pages while enabling the proxy. Table VI shows a summary of the FP evaluation for three threshold values (t = 7.19*10^{-2}, t = 1.0*10^{-2}, t = 0.002) for the three applications. The last row shows the overall number of benign inputs, total number of false warning along with FP rate.

Table V shows the performance of XSS attack detection based on KLD for various threshold values. Columns 2–5 show the number of attack inputs that would be missed if we set the threshold (t) values as 0.05, 0.5, 1.0, and 2.0, respectively. The last column shows the total number of attack inputs supplied during the evaluation. The final rows accounts for the total number of undetected attack along with false negative (FN) rate (# of undetected attack/# of total attack). We notice that when threshold is 0.05, all the attacks are detected for all three applications (i.e., 0% FN). However, as the value of t increases, applications under XSS attacks may go undetected by our proxy-based approach. The total numbers of undetected attack are 2, 9, and 13, for the threshold values 0.5, 1.0, and 2.0, respectively. The highest FN occurs when t is set as 2.0 (13%). These observations provide us an insight of the appropriate threshold value (should be close to zero).

<table>
<thead>
<tr>
<th>Application</th>
<th>OSVDB ID</th>
<th>$a_0$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>92537</td>
<td>14.08</td>
<td>0.81</td>
<td>12.36</td>
<td>0.64</td>
<td>7.59</td>
</tr>
<tr>
<td>81984</td>
<td>8.23</td>
<td>3.24</td>
<td>7.82</td>
<td>5.93</td>
<td>6.71</td>
<td></td>
</tr>
<tr>
<td>81986</td>
<td>11.20</td>
<td>1.11</td>
<td>12.02</td>
<td>0.35</td>
<td>13.54</td>
<td></td>
</tr>
<tr>
<td>81987</td>
<td>8.79</td>
<td>8.02</td>
<td>2.70</td>
<td>9.83</td>
<td>10.70</td>
<td></td>
</tr>
<tr>
<td>81985</td>
<td>3.37</td>
<td>2.31</td>
<td>3.25</td>
<td>10.43</td>
<td>5.30</td>
<td></td>
</tr>
<tr>
<td>80833</td>
<td>7.67</td>
<td>8.92</td>
<td>0.71</td>
<td>10.14</td>
<td>12.88</td>
<td></td>
</tr>
<tr>
<td>45966</td>
<td>15.55</td>
<td>11.62</td>
<td>14.88</td>
<td>9.25</td>
<td>15.55</td>
<td></td>
</tr>
<tr>
<td>53389</td>
<td>2.32</td>
<td>14.08</td>
<td>13.92</td>
<td>15.01</td>
<td>9.64</td>
<td></td>
</tr>
<tr>
<td>43668</td>
<td>15.04</td>
<td>8.49</td>
<td>8.88</td>
<td>9.03</td>
<td>13.02</td>
<td></td>
</tr>
<tr>
<td>Serendipity</td>
<td>95176</td>
<td>3.22</td>
<td>11.44</td>
<td>0.60</td>
<td>14.83</td>
<td>12.39</td>
</tr>
<tr>
<td>87395</td>
<td>15.93</td>
<td>8.30</td>
<td>2.65</td>
<td>12.02</td>
<td>9.16</td>
<td></td>
</tr>
<tr>
<td>81713</td>
<td>8.27</td>
<td>2.57</td>
<td>4.23</td>
<td>3.18</td>
<td>6.74</td>
<td></td>
</tr>
<tr>
<td>76836</td>
<td>11.72</td>
<td>3.47</td>
<td>13.21</td>
<td>6.39</td>
<td>3.65</td>
<td></td>
</tr>
<tr>
<td>76856</td>
<td>10.75</td>
<td>5.21</td>
<td>0.21</td>
<td>14.09</td>
<td>10.41</td>
<td></td>
</tr>
<tr>
<td>75777</td>
<td>9.41</td>
<td>3.52</td>
<td>3.15</td>
<td>15.51</td>
<td>14.76</td>
<td></td>
</tr>
<tr>
<td>75484</td>
<td>6.36</td>
<td>8.75</td>
<td>13.89</td>
<td>2.89</td>
<td>6.56</td>
<td></td>
</tr>
<tr>
<td>72726</td>
<td>8.37</td>
<td>12.60</td>
<td>14.61</td>
<td>14.82</td>
<td>14.13</td>
<td></td>
</tr>
<tr>
<td>67693</td>
<td>13.17</td>
<td>5.92</td>
<td>5.14</td>
<td>10.33</td>
<td>14.16</td>
<td></td>
</tr>
<tr>
<td>68944</td>
<td>5.95</td>
<td>0.02</td>
<td>2.91</td>
<td>4.86</td>
<td>3.41</td>
<td></td>
</tr>
<tr>
<td>26999</td>
<td>10.16</td>
<td>6.66</td>
<td>8.01</td>
<td>2.24</td>
<td>14.05</td>
<td></td>
</tr>
<tr>
<td>PHP fusions</td>
<td>90697</td>
<td>9.32</td>
<td>5.32</td>
<td>9.91</td>
<td>12.52</td>
<td>11.86</td>
</tr>
<tr>
<td>90078</td>
<td>2.08</td>
<td>4.85</td>
<td>6.11</td>
<td>6.49</td>
<td>2.81</td>
<td></td>
</tr>
<tr>
<td>90707</td>
<td>2.09</td>
<td>2.14</td>
<td>4.40</td>
<td>5.17</td>
<td>5.55</td>
<td></td>
</tr>
<tr>
<td>90706</td>
<td>8.87</td>
<td>3.15</td>
<td>6.44</td>
<td>15.72</td>
<td>15.37</td>
<td></td>
</tr>
<tr>
<td>90705</td>
<td>13.99</td>
<td>1.61</td>
<td>1.07</td>
<td>8.67</td>
<td>12.66</td>
<td></td>
</tr>
<tr>
<td>90704</td>
<td>0.03</td>
<td>3.68</td>
<td>3.12</td>
<td>3.86</td>
<td>12.47</td>
<td></td>
</tr>
<tr>
<td>90703</td>
<td>12.81</td>
<td>0.87</td>
<td>1.58</td>
<td>15.48</td>
<td>13.26</td>
<td></td>
</tr>
<tr>
<td>90702</td>
<td>14.55</td>
<td>14.14</td>
<td>9.07</td>
<td>5.65</td>
<td>8.89</td>
<td></td>
</tr>
</tbody>
</table>

Table VI shows a summary of the FP rates for three threshold values (t = 7.19*10^{-2}, t = 1.0*10^{-2}, t = 0.002) for the three applications. The last row shows the overall number of benign inputs, total number of false warning along with FP rate.

Table VI. Summary of False-Positive (FP) Rate for Various Threshold Levels

- **PHP-Address book**: 108 inputs, 2 false positives (FP), 18%.
- **Serendipity**: 132 inputs, 3 FP, 2.2%.
- **PHP fusions**: 204 inputs, 4 FP, 1.9%.

We apply total 108, 132, and 204 benign inputs for **PHP Address book**, **Serendipity**, and **PHP fusions**, respectively. The FP rates ranges between 1.8% and 4.6%, 1.5% and 3.0%, and 1.9% and 3.4% for **PHP Address book**, **Serendipity**, and **PHP fusions**, respectively. The highest FP rate is observed for the **PHP Address-book** (4.6%, for t = 0.002) and the lowest FP rate is found for **Serendipity** (1.5%, t = 5.19*10^{-2}). The overall FP rate is between 2.0% and 3.6%. Our observation indicates that setting the threshold value (t) close to zero results in avoiding false positive warnings. This allows us to minimize the FN as well.
V. CONCLUSIONS

This paper develops a proxy level XSS attack detection approach based on Kullback-Leibler Divergence (KLD) metric. Our approach relies on distance between the probability distribution of legitimate JavaScript code and the observed JavaScript code present in a response page. The deviation between the two types of JavaScript results in high KLD value. The KLD-based XSS attack detection approach relies on constant back-off smoothing technique. We apply the approach to test three vulnerable PHP applications. The evaluation results indicate that the approach can successfully detect widely known XSS attack signatures by setting appropriate threshold values. The estimated FN rates vary between 0% and 9.2%, and the FP rates vary between 1.5% and 4.6%. We show a comparative analysis for designer to choose the most appropriate threshold values to minimize FP and FN rates. Our future work plan includes evaluating the approach for more web applications, applying KLD based on other smoothing techniques, and detecting attacks in presence of external JavaScript files.

REFERENCES