Information Theoretic XSS Attack Detection in Web Applications

Hossain Shahriar¹, Sarah North¹, Wei-Chuen Chen², and Edward Mawangi²
Department of Computer Science
Kennesaw State University
Kennesaw, Georgia 30144 USA
¹{hshahria, snorth}@kennesaw.edu, ²{wchen10, emwangi8}@students.kennesaw.edu

Abstract—Cross-Site Scripting (XSS) has been ranked among the top three vulnerabilities over the last few
years. XSS vulnerability allows an attacker to inject arbitrary JavaScript code that can be executed in the
victim’s browser to cause unwanted behaviors and security breaches. Despite the presence of many
mitigation approaches, the discovery of XSS is still widespread among today’s web applications. As a
result, there is a need to improve existing solutions and to develop novel attack detection techniques. This
paper proposes a proxy-level XSS attack detection approach based on a popular information-theoretic
measure known as Kullback-Leibler Divergence (KLD). Our intuition is that legitimate JavaScript code
present in an application should remain similar or very close to the JavaScript code present in a rendered
web page. A deviation between the two can be an indication of an XSS attack. We apply a back-off
smoothing technique to effectively detect the presence of malicious JavaScript code in response pages. 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 through proper choice of threshold values of KLD. Further, the performance
overhead has been found to be negligible.

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

INTRODUCTION

Vulnerabilities are frequently discovered in web applications. Among all known vulnerabilities, Cross-Site
Scripting (XSS) has been ranked among the top three vulnerabilities over the last few years (OWASP 2013). XSS
vulnerability opens up the possibility for an attacker to inject arbitrary JavaScript code (OWASP-XSS 2013) that
can execute in the context of a victim’s browser. The injected script code causes unwanted behaviors (e.g.,
generating pop up windows) and security breaches (e.g., session hijacking (Msujaws 2011)). A recent survey also
shows that on average 60% or more web applications are currently suffering from XSS vulnerabilities (Tudor
2013). Given that statistic, addressing the mitigation of XSS vulnerabilities is important.

Despite the presence of many mitigation approaches for XSS attacks at both client and server sides (Shar et al.
2009), the discovery of XSS vulnerability is still widespread among today’s web applications. Most of these
approaches rely on signature-based attack detection that are effective in detecting known attack symptoms. Thus,
there is a need to develop anomaly-based attack detection techniques that may detect unknown and new attack
signatures. This paper applies an information theoretic concept to detect XSS attacks. Further, very few works have
explored detecting XSS attacks at the proxy level.

In this paper, we propose a proxy-level XSS attack detection technique based on a popular information theoretic
measure known as Kullback-Leibler Divergence (KLD)¹. 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 set of JavaScript code may indicate XSS attacks. Our contribution remains in addressing
the missing elements when computing KLD between the set of expected and actual JavaScript code. In particular,
we apply the constant back-off smoothing technique that we brought from information retrieval literature.

¹ An earlier version of this article has been published in Shahriar et al. 2013.
We apply the proposed XSS attack detection approach for web applications implemented in PHP language and containing known XSS vulnerabilities. The initial results show that the approach can detect most of the known XSS attack signatures and show negligible false positive warning. Further, it imposes negligible runtime overhead. The proposed approach can handle diverse types of JavaScript code commonly found in web applications such as inline, URL attribute, and Cascading Style Sheet (CSS). Moreover, it can be applied as a complementary defense technique for applications that may lack an adequate XSS input filtering mechanism.

This paper is organized as follows: First, we show an example of XSS attack followed by a brief introduction of related work. Then the proposed KLD-based XSS attack detection framework is discussed along with a working example. We then discuss the experimental results. Finally, we conclude the paper and discuss future work.

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 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. As a result, if a malicious user supplies arbitrary JavaScript code, it would execute when the response page is displayed in the victim's browser.

![HTML FORM](A) | ![PHP SCRIPT (SHOW.PHP)](B)
--- | ---
<form action = "show.php" method= "post">  
<input type = "text" name = "user" value = ">
<input type = "submit" name = "submit" value = "Submit">
</form> | Hello <?php  
echo $_POST["user"];  
?>

Figure 1. HTML code and XSS vulnerable PHP code

For example, if the user name is supplied as `<script>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 attack payloads are retrieved at a later time when a victim sends a page request at the client side.

Note that XSS attacks bypass the default Same Original Policy (SOP) in browsers which is intended to prevent accessing one webpage downloaded from a domain to a page from another domain. For example, a webpage downloaded from http://www.goodsite.com might include a script tag `<script src="http://www.badsite.org/x.js"></script>`. The script tag results in downloading the x.js script file from http://www.badsite.com and the JavaScript code gets executed by browsers in the context of http://www.goodwebsite.com. Browsers cannot detect which part of JavaScript code is injected or unwanted.

RELATED WORK

Related work on XSS detection

XSS attack has been detected at both the server and client-sides. Shar et al. 2012 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. 2012 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 Jim et al. 2007 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. 2009 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. 2009 encode all legitimate JavaScript function calls as syntactically invalid code so that attacker injected code gets executed. Nadji et al. 2009 detect XSS attacks through the notion of document structure integrity validation where expected and actual parse trees of DOMs are compared. Johns et al. 2008 detect XSS attacks by analyzing requests

Several works apply dynamic analysis at the server side (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 (Pietraszek et al. 2005; Futoransky et al. 2007; Chin et al. 2009) 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 interpreters so that program operations become taint-aware.

A number of works propose XSS attack detection techniques at the client-side. Kirda et al. 2006 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. 2009 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 (Shahriar et al. 2008). Interested readers can see (Shahriar et al. 2012) 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 (W3af 2013, Acutenix 2013). 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.

**Related work on Kullback-Leibler Divergence (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 (Briggs 2003), masquerade attack detection (Tapiador et al. 2010, and outlier data value detection in wireless sensor network (Li et al. 2012).

Briggs 2003 applies KLD to identify authorship of documents. They first build model of each author by aggregating documents generated by the author. They first develop a set of candidate models. 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. 2010 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. 2012 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. 2010 measures the similarity of events based on KLD and applied it in the domain of fuel-cell study. Sarkar et al. 2007 apply information theoretic measure including KLD to measure the quality of modularization in non-object oriented software systems.

**Proxy-level detection of XSS attacks**

In this section, we first briefly introduce the proxy-level framework for XSS attack detection. We then discuss the KLD measurement formula and the related properties. Then we highlight the necessity of smoothing technique followed by the algorithm for computing KLD with constant back-off smoothing technique. Finally, we show an example of XSS attack detection using KLD with smoothing technique.
Framework

Figure 2 shows a diagram of the proposed proxy-level attack detection approach. In particular, the XSS Detector module (we denote it as a 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 JavaScript code \( Q \). 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 retrieves \( P \) (legitimate set of JavaScript code that is learned from application source code beforehand) 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 \( Q \) is pre-computed for each of the server side pages. In particular, it requires identifying all JavaScript code from the server side script pages followed by tokenizing them (based on common separator such as \(<, >, ;\) to identify unique elements and their occurrence probability. Our current approach is scalable for large amount of JavaScript code. Currently, in can handle all inline JavaScript code (e.g., \(<\text{script}>\ldots</\text{script}>\), code present as URL attribute (e.g., \(<\text{img} \text{ src= "javascript:…">}) and CSS (e.g., \(<\text{div} \text{ style= "background:url('javascript:…')">})). 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 \( s1 \) with probability 1 in \( P \{s1:1\} \). This is to avoid having an empty \( P \) set, which makes it impractical for computing the distance between \( P \) and any other distribution.

\[
\text{Figure 2. Proxy-level framework for testing XSS.}
\]

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 \( P = \{p_1, \ldots, p_n\} \) and \( Q = \{q_1, \ldots, q_n\} \). Then, the KLD is defined as follows (Cover & Thomas, 2006):

\[
\text{KLD}(P, Q) = \sum_{i=1}^{n} p_i \ast \log_2 \left(\frac{p_i}{q_i}\right) \ldots (i)
\]

Here, the following two constraints (Equations (ii) and (iii)) are satisfied:

\[
\sum_{i=1}^{n} p_i = 1 \ldots (ii)
\]

\[
\sum_{i=1}^{n} q_i = 1 \ldots (iii)
\]

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 (\( P \)). Therefore, KLD is also denoted as the relative entropy between \( P \) and \( Q \) in information theory. Note that KLD is not symmetric (i.e., \( \text{KLD}(P, Q) \neq \text{KLD}(Q, P) \)). Also, \( \text{KLD}(P, Q) = 0 \), 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 \( p_i \ast \log_2 \left(\frac{p_i}{q_i}\right) \) can be rewritten as subtraction of two terms: \( p_i \ast \log_2(p_i) - p_i \ast \log_2(q_i) \). While we compute \( \text{KLD}(P, Q) \), two cases might arise:

(a) Case 1: \( i^{th} \) element is present in \( P \) \((p_i \neq 0)\), but not in \( Q \) \((q_i = 0)\). As a result, \( \log_2(q_i) \) becomes \( \log_2(0) \), which is infinite. Therefore, the term \( p_i \ast \log_2(p_i/q_i) \) becomes infinite. This results in an infinite distance between \( P \) and \( Q \).

(b) Case 2: \( i^{th} \) element is not present in \( P \) \((p_i = 0)\), but present in \( Q \) \((q_i \neq 0)\). Similar to Case 1, we can show that \( p_i \ast \log_2(p_i/q_i) \) is infinite, and hence \( \text{KLD}(P, Q) \) becomes infinite.
In the literature, smoothing technique (Briggs 2003, Ney et al. 1994; Zhai 2002; Jelinek 1980] is applied to avoid having infinite distance between two probability distributions. Among a few well-known techniques, we choose to apply the constant back-off smoothing technique (Ney et al. 1994). Applications of other smoothing techniques may bring similar XSS detection results, and is left as part of the future work.

**Computing KLD with smoothed probability distributions**

We apply a back-off (or absolute discounting-based) smoothing to compute the similarity between a known set of JavaScript implemented by a programmer ($P$) and the actual observed JavaScript code ($Q$) for a given page at the proxy-level. The JavaScript code is tokenized to identify unique elements. Here, the elements that are missing in either $P$ or $Q$ sets are added with a very negligible amount of constant probability in the respective set. At the same time, the present elements in these two sets are subtracted with constant probability values in such 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, c)](algorithm1.png)

Figure 3. Algorithm for computing KLD based on smoothing.

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 unseen 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)](algorithm2.png)

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 $e$. 
An example of XSS attack detection based on KLD

We now show an example of XSS attack detection using KLD computation. We reuse the PHP code example shown in Figure 1(b) where we substitute the PHP variable ($POST["user"] ) with an injected JavaScript code (<script>alert(document.cookie);</script>). Note that the original program code does not have any JavaScript code, so we have P = {s1}.

<table>
<thead>
<tr>
<th>Input value ($POST[&quot;user&quot;] )</th>
<th>Response page</th>
<th>Distribution (Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;script&gt; alert (document.cookie); &lt;/script&gt;</td>
<td>Hello &lt;script&gt;alert (document.cookie);&lt;/script&gt;</td>
<td>{script: 2/5, alert: 1/5, document: 1/5, cookie: 1/5}</td>
</tr>
</tbody>
</table>

Table I shows the response page (2nd column) and the corresponding token obtained from observed JavaScript code Q = {script: 2/5, alert: 1/5, document: 1/5, cookie: 1/5}. We apply Algorithm 1 (Figure 3) to generate smoothed population P’ and Q’ from P and Q, respectively as follows:

CP = 1, CQ = 4; SU = {s1, script, alert, document, cookie}
P’ = {s1: 1-e, script:e/4, alert:e/4, document:e/4, cookie:e/4}
Q’ = {s1:e, script:2/5-e/4, alert:1/5-e/4, document:1/5-e/4, cookie:1/5-e/4}

Table II shows the detailed computation of KLD between P’ and Q’ based on each of the elements observed in both sets. The value generated in the last column is summed up to find the final KLD. If we assume that the threshold value to compare this KLD is 0, then an attack will be detected.

The KLD computation 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.

EXPERIMENTAL EVALUATION

We implement the proposed proxy-level XSS detection approach by extending the Fiddler (Fiddler 2013) tool. It is a debugging tool for web applications. It can intercept request and response pages. In our case, we implement the necessary Java code to analyze the response page. The server side of the web application is lightly instrumented to build legitimate JavaScript code and develop 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 and contact manager), Serendipity (blog management), and PHP-fusions (content management system). In Table III we show 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).

<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>
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-Cheat Sheet 2013. Table IV shows the description of each type of XSS attack along with example payload (a0-a4). The attacks include injecting JavaScript through HTML attributes and invoking onload (a0), onmouseover (a1), onerror (a2) method calls, encoded URI (a3), and code encoding (a4).

<table>
<thead>
<tr>
<th>Description</th>
<th>Attack payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS in Attributes – onload (a0)</td>
<td><code>&lt;body onload=alert('test1')&gt;</code></td>
</tr>
<tr>
<td>XSS in Attributes – onmouseover (a1)</td>
<td><code>&lt;b onmouseover = alert('Wuffff!')&gt;</code> click me!&lt;/b&gt;</td>
</tr>
<tr>
<td>XSS in Attributes- onerror (a2)</td>
<td><code>&lt;img src=&quot;http://url.to.file.which/not.exist&quot; onerror = alert (document.cookie);&gt;</code></td>
</tr>
<tr>
<td>XSS via Encoded URI (a3)</td>
<td><code>&lt;IMG SRC=jscript:alert('test2')&gt;</code></td>
</tr>
<tr>
<td>XSS using code encoding (a4)</td>
<td><code>&lt;META HTTP-EQUIV=&quot;refresh&quot; CONTENT=&quot;0;url = data:text/html;base64,PHNjcmlwdD5hbGVydG9yc29uDx%2F%7B%7 Chanel=sv8C%7D&quot;&gt;</code></td>
</tr>
</tbody>
</table>

To identify vulnerable input locations, we rely on the reported XSS vulnerabilities from OSVDB. Table V shows the list of OSVDB ID that we examine to identify the responsible pages that can be used to inject arbitrary JavaScript code. We identify the relevant pages that allow viewing/rendering the injected code. While enabling the proxy-level detector, we examine whether the injected code can be flagged as malicious by the proxy or not to measure the effectiveness of our approach.

<table>
<thead>
<tr>
<th>Application</th>
<th>OSVDB ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>92537, 81984, 81986, 81987, 81985, 80833, 45966, 53389, 43668</td>
</tr>
<tr>
<td>Serendipity</td>
<td>95176, 87395, 81713, 76836, 76856, 75777, 75484, 72726, 67693, 68944, 26999</td>
</tr>
<tr>
<td>PHP-fusions</td>
<td>90697, 90708, 90707, 90706, 90705, 90704, 90703, 90702, 90701, 90700, 90699, 90698, 80680, 87961, 75485, 51053, 36342</td>
</tr>
</tbody>
</table>

Table VI 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 a0-a4. Figure 5 shows the KLD values for attacks a0-a4 (after sorting all KLD values column wise and excluding the OSVDB ID column). The lowest and highest KLD values we obtain 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). For a0, a1, a2, a3, and a4, the lowest and highest KLD were 0.03 and 15.93, 0.02 and 14.14, 0.21 and 14.88, 0.35 and 15.72, and 2.81 and 15.55. The KLD value level depends on the present JavaScript code in the vulnerable script files. It also depends on the supplied injected attack script code. Note during this evaluation, the constant probability (e) used for back-off smoothing algorithm was 0.0001.

Table VII 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).

---

2 We considered vulnerability reported between the years 2007-2013.
TABLE VI.  KLD VALUES DIFFERENT XSS ATTACK TYPES

<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></td>
<td>81984</td>
<td>8.23</td>
<td>3.24</td>
<td>7.82</td>
<td>5.93</td>
<td>6.71</td>
</tr>
<tr>
<td></td>
<td>81986</td>
<td>11.20</td>
<td>1.11</td>
<td>12.02</td>
<td>0.35</td>
<td>13.54</td>
</tr>
<tr>
<td></td>
<td>81987</td>
<td>8.79</td>
<td>8.02</td>
<td>2.70</td>
<td>9.83</td>
<td>10.70</td>
</tr>
<tr>
<td></td>
<td>81985</td>
<td>3.37</td>
<td>2.31</td>
<td>3.25</td>
<td>10.43</td>
<td>5.30</td>
</tr>
<tr>
<td></td>
<td>80833</td>
<td>7.67</td>
<td>8.92</td>
<td>0.71</td>
<td>10.14</td>
<td>12.88</td>
</tr>
<tr>
<td></td>
<td>45966</td>
<td>15.55</td>
<td>11.62</td>
<td>14.88</td>
<td>9.25</td>
<td>15.55</td>
</tr>
<tr>
<td></td>
<td>53389</td>
<td>2.32</td>
<td>14.08</td>
<td>13.92</td>
<td>15.01</td>
<td>9.64</td>
</tr>
<tr>
<td></td>
<td>43668</td>
<td>15.04</td>
<td>8.49</td>
<td>8.88</td>
<td>9.03</td>
<td>13.02</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></td>
<td>87395</td>
<td>15.93</td>
<td>8.30</td>
<td>2.65</td>
<td>12.02</td>
<td>9.16</td>
</tr>
<tr>
<td></td>
<td>81713</td>
<td>8.27</td>
<td>2.57</td>
<td>4.23</td>
<td>3.18</td>
<td>6.74</td>
</tr>
<tr>
<td></td>
<td>76836</td>
<td>11.72</td>
<td>3.47</td>
<td>13.21</td>
<td>6.39</td>
<td>3.65</td>
</tr>
<tr>
<td></td>
<td>76856</td>
<td>10.75</td>
<td>3.21</td>
<td>0.21</td>
<td>14.09</td>
<td>10.41</td>
</tr>
<tr>
<td></td>
<td>75777</td>
<td>9.41</td>
<td>3.52</td>
<td>3.15</td>
<td>15.51</td>
<td>14.76</td>
</tr>
<tr>
<td></td>
<td>75484</td>
<td>6.36</td>
<td>8.75</td>
<td>13.89</td>
<td>2.89</td>
<td>6.56</td>
</tr>
<tr>
<td></td>
<td>72726</td>
<td>8.37</td>
<td>12.60</td>
<td>14.61</td>
<td>14.82</td>
<td>14.13</td>
</tr>
<tr>
<td></td>
<td>67693</td>
<td>13.17</td>
<td>5.92</td>
<td>5.14</td>
<td>10.33</td>
<td>14.16</td>
</tr>
<tr>
<td></td>
<td>68944</td>
<td>5.95</td>
<td>0.02</td>
<td>2.91</td>
<td>4.86</td>
<td>3.41</td>
</tr>
<tr>
<td></td>
<td>26999</td>
<td>10.16</td>
<td>6.66</td>
<td>8.01</td>
<td>2.24</td>
<td>14.05</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></td>
<td>90708</td>
<td>2.08</td>
<td>4.85</td>
<td>6.11</td>
<td>6.49</td>
<td>2.81</td>
</tr>
<tr>
<td></td>
<td>90707</td>
<td>2.09</td>
<td>2.14</td>
<td>4.40</td>
<td>5.17</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>90706</td>
<td>8.87</td>
<td>3.15</td>
<td>6.44</td>
<td>15.72</td>
<td>15.37</td>
</tr>
<tr>
<td></td>
<td>90705</td>
<td>13.99</td>
<td>1.61</td>
<td>1.07</td>
<td>8.67</td>
<td>12.66</td>
</tr>
<tr>
<td></td>
<td>90704</td>
<td>0.03</td>
<td>3.68</td>
<td>3.12</td>
<td>3.86</td>
<td>12.47</td>
</tr>
<tr>
<td></td>
<td>90703</td>
<td>12.81</td>
<td>0.87</td>
<td>1.58</td>
<td>15.48</td>
<td>13.26</td>
</tr>
<tr>
<td></td>
<td>90702</td>
<td>14.55</td>
<td>14.14</td>
<td>9.07</td>
<td>5.65</td>
<td>8.89</td>
</tr>
</tbody>
</table>

Figure 5.  KLD values for $a_0$-$a_4$

TABLE VII.  XSS DETECTION EFFECTIVENESS FOR VARIOUS THRESHOLD VALUES

<table>
<thead>
<tr>
<th>Application</th>
<th>t=0.05</th>
<th>t=0.5</th>
<th>t=1.0</th>
<th>t=2.0</th>
<th>Total # of attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>Serendipity</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>55</td>
</tr>
<tr>
<td>PHP fusions</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td># of undetected attack (FN%)</td>
<td>0 (0%)</td>
<td>2 (1.4%)</td>
<td>9 (6.4%)</td>
<td>13 (9.2%)</td>
<td>140</td>
</tr>
</tbody>
</table>
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 VIII shows a summary of the FP evaluation for three threshold values ($t = 7.19 \times 10^{-7}$, $t = 5.19 \times 10^{-2}$, $t = 0.002$) for the three applications. The last row shows the overall number of benign inputs, total number of false warning along with FP rate.

**TABLE VIII. EVALUATION SUMMARY OF FALSE POSITIVE (FP) RATE FOR VARIOUS THRESHOLD LEVELS**

<table>
<thead>
<tr>
<th>Application</th>
<th>Total # of benign inputs</th>
<th>$t = 7.19 \times 10^{-7}$ (FP%)</th>
<th>$t = 5.19 \times 10^{-2}$ (FP%)</th>
<th>$t = 0.002$ (FP%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>108</td>
<td>2 (1.8%)</td>
<td>4 (3.7%)</td>
<td>5 (4.6%)</td>
</tr>
<tr>
<td>Serendipity</td>
<td>132</td>
<td>3 (2.2%)</td>
<td>2 (1.5%)</td>
<td>4 (3.0%)</td>
</tr>
<tr>
<td>PHP fusions</td>
<td>204</td>
<td>4 (1.9%)</td>
<td>5 (2.4%)</td>
<td>7 (3.4%)</td>
</tr>
<tr>
<td># of false alarm (FP%)</td>
<td>444</td>
<td>9 (2.0%)</td>
<td>11 (2.5%)</td>
<td>16 (3.6%)</td>
</tr>
</tbody>
</table>

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%, $t = 5.19 \times 10^{-2}$). 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. Also by comparing the findings from Tables VII (FN rate) and VIII (FP rate), we suggest to set the threshold value $t$ that minimizes FP. This would allow us to minimize the FN as well while detecting XSS attacks based on KLD.

Finally, we assess the overhead imposed by the proxy-based XSS detection approach. Notably, we measure the time delay required to analyze response page which subsequently affect the rendering of a web page at the client side. Table IX shows the average delay to receive a response page during our evaluation. We compute the delay by averaging the delays incurred due to the analysis of all response pages during our evaluation (including pages generated due to both malicious and benign inputs). We find that *PHP Fusions* has the highest average delay (412ms), followed by the *PHP Address Book* (354ms) and *Serendipity* (233ms). The major reason of having the delay is to extract and tokenize JavaScript code, as well as computing KLD based on smoothed distributions. Some the response pages that have large amount of inline JavaScript code require more time to process by the proxy.

**TABLE IX. PERFORMANCE OVERHEAD EVALUATION**

<table>
<thead>
<tr>
<th>Application</th>
<th>Delay (millisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>354</td>
</tr>
<tr>
<td>Serendipity</td>
<td>233</td>
</tr>
<tr>
<td>PHP fusions</td>
<td>412</td>
</tr>
</tbody>
</table>

**CONCLUSIONS**

XSS is one of the most common vulnerabilities discovered in web applications. This paper develops a proxy-level XSS attack detection approach based on Kullback-Leibler Divergence (KLD). Our approach relies on distance between the probability distribution of legitimate JavaScript code and the observed JavaScript code present in a response page. A proxy-level framework has been developed to compute KLD to identify the distance between the two types of JavaScript code. The deviation between the two types of JavaScript results in a KLD value greater than zero. The KLD-based XSS detection approach also relies on constant back-off smoothing technique of probability distribution to consider the unseen JavaScript tokens among the two sets of code distributions. We implement the approach by extending a web debugging tool and 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 also show a comparative analysis for designer to choose
the most appropriate threshold values to minimize FP and FN rates. The approach imposes negligible overhead at the proxy-level.

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 which are part of legitimate web applications.

ACKNOWLEDGEMENT

This work is supported in part by STEM/LSAMP (Louis Stokes Alliances for Minority Participation) Scholarship Program.

REFERENCES


