Server-side code injection attack detection based on Kullback-Leibler distance

Hossain Shahriar*, Sarah M. North, YoonJi Lee and Roger Hu

Department of Computer Science,
Kennesaw State University,
Kennesaw, Georgia 30144, USA
E-mail: hshahria@kennesaw.edu
E-mail: snorth@kennesaw.edu
E-mail: ylee31@students.kennesaw.edu
E-mail: rhu1@students.kennesaw.edu
*Corresponding author

Abstract: In this paper, we apply a well-known measure from information theory domain called Kullback-Leibler distance (or divergence) (KLD) to detect the symptoms of code injection attacks early during programme runtime. We take advantage of the observation that during code injection attack, the intended structure deviates from the expected structure. The KLD can be a suitable measure to capture the deviation. Our contribution includes the development of a server-side framework to compute KLD. In particular, we apply a smoothing algorithm to avoid the infinite KLD distance during attack detection stage. We evaluate our approach with three PHP applications having SQLI and XSS vulnerabilities. The initial results show that KLD can be an effective measurement technique to detect the occurrence of code injection attacks. The approach suffers from lower false positive and negative rates, and imposes negligible runtime overhead.

Keywords: Kullback-Leibler distance; KLD; code injection attack; web application security; information theory; cross-site scripting; XSS; SQL injection; server-side attack detection.


Biographical notes: Hossain Shahriar is currently an Assistant Professor of Computer Science at Kennesaw State University, Georgia, USA. His research interests include web and mobile application security vulnerabilities and their mitigation techniques. He has published research articles in many international journals and conferences. His research has attracted a number of awards including the Best Paper Award in IEEE DASC, Outstanding Research Achievement Award from Queen’s University, Canada, and IEEE Kingston Section Research Excellence Award. He has been a reviewer of many international journals (Computers and Security, Journal of System and Software, and ACM Computing Surveys) and served PC member of international conferences on software, computer, and application security (IEEE COMPASC, IEEE ITNG, ACM SAC, ACM/SIGSAC SIN).

Copyright © 200x Inderscience Enterprises Ltd.
Currently, he is a member of the ACM, ACM SIGAPP, and IEEE. More information about his research and background can be found at http://cs.kennesaw.edu/hshahria/.

Sarah M. North is an Assistant Professor of Computer Science Department at Kennesaw State University. She has been successfully involved in the research in the areas of information security education, human-computer interaction and cognitive science. She has published several book chapters; and a number of referred scholarly journals and conferences including articles in national and international venues such as ACM/SIGHCI, IEEE HASE, and IJNSA. She also served as a Principal/Co-principal Investigator on a number of research grants sponsored by the Boeing Company, National Science Foundation, and National Security Agency. She is a member of the ACM and IEEE. More information about her research/publications and background can be found at http://cs.kennesaw.edu/snorth.

YoonJi Lee is currently pursuing her undergraduate Computer Science minor in Information Security and Assurance degree programme at Kennesaw State University. Her research interest is in Android application, security in computing and vulnerability assessment. She completed her military duty in 2013. She is a member of Science, Technology, Engineering, and Mathematics (STEM), Association for Computing Machinery (ACM) and a recipient of NASA Scholarship.

Roger Hu is currently pursuing his BSCS degree at Kennesaw State University. His current research interests include web and mobile application security. He is also a STEM scholar mentored by Drs. Shahriar and North. He is serving as the President of KSU CS ACM Student Chapter. His other interests include participating and volunteering for ACM ICPC programming competition.

This paper is a revised and expanded version of a paper entitled ‘Design and development of anti-XSS proxy’ presented at the 8th IEEE Internet Technology and Secured Transactions (ICITST), London, UK, 9–11 December 2013.

1 Introduction

We rely on web applications to perform various activities on a daily basis. Unfortunately, most web applications are vulnerable to code injection attacks. A recent survey from OWASP (2014) indicates that SQL injection (OWASP-SQLI, 2014) and cross-site scripting (XSS) (OWASP-XSS, 2014) are the two worst code injection vulnerabilities commonly discovered. These vulnerabilities have been ranked among top three over the last few years. These two vulnerabilities can be exploited to perform many unwanted activities. For example, SQLI can be applied to bypass authentication, alter or even delete information being processing by applications (Halfond et al., 2006). A recent report (Paganinip, 2013) indicates that SQLI attacks are now conducted using trusted search engine bots (Google and Bing). Similarly, XSS vulnerability opens up the possibility for an attacker to inject arbitrary HTML content (OWASP-XSS, 2014) that may alter the display of a web page. XSS can also be applied to inject arbitrary JavaScript code that can execute in the context of a victim’s browser and perform unwanted actions such as reading session or cookie and passing to unwanted third party websites (Msujaws, 2011).
Given that it is important to detect the exploitation of vulnerabilities to prevent some of these unwanted consequences.

Many literature work has addressed the mitigation of SQLI (e.g., Halfond and Orso, 2005; Boyd and Keromytis, 2004; Thomas and Williams, 2007; Kemalis and Tzouramanis, 2008) and XSS (Shar and Tan, 2012; Frenz and Yoon, 2012; Jim et al., 2007; Frenz and Yoon, 2012; Iha and Doi, 2009; Wurzinger et al., 2009). However, the number of discovered code injection vulnerabilities is still on the rise (OSVDB, 2014). Thus, new mitigation technique development is needed. Further, most of the works rely on signature-based attack detection schemes 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.

In this paper, we propose a code injection attack detection technique based on popular information-theoretical measure namely Kullback-Leibler distance (or divergence) (KLD).\(^1\) We take advantage of the observation that during code injection attack, the intended structure (e.g., SQL query for SQLI, HTML response page for XSS) deviates from the expected structure. The KLD can be a suitable measure to understand how much alteration or deviation occurs at runtime. Our contribution remains in the development of a server-side framework to compute KLD for attack detection. Moreover, we also apply the constant back-off smoothing algorithm to avoid the infinite KLD among two structures during attack detection. We evaluate the proposed approach with three open source PHP applications having SQLI and XSS vulnerabilities. The initial results show that KLD is an effective measure to detect against both SQLI and XSS attacks. Further, the approach suffers from lower false positive (FP) and negative rates, and imposes negligible overhead.

This paper is organised as follows: Section 2 shows an example of SQLI and XSS attack. Section 3 discusses the related work to mitigate these two attacks. We also discuss related works that apply KLD to solve various problems including security. Section 4 discusses our proposed framework for code injection attack detection followed by examples of KLD-based attack detection. Section 5 describes the experimental evaluation results. Finally, Section 6 concludes this paper and discusses future work.

2 Background and related work

2.1 An example of SQLI attack

We provide an example of an SQLI attack based on a PHP code snippet shown in Figure 1. Lines 1 and 2 extract user supplied information from Login and Password fields into $login and $pwd variables, respectively. The inputs are not filtered and concatenated at line 3 to generate a dynamic SQL query. Line 4 executes the query and lines 5 and 6 perform the authentication. If there is at least one row present in the result set, then a session id is established. Let us assume that a malicious hacker provides the value of $login as ‘or 1=1 --’ while leaving the second input variable ($pwd) as blank. The resultant query becomes “select id, level from tlogin where login = ‘or 1=1 --’ and password = ‘”’. Since anything after – is considered as comment, the effective query becomes “select id, level from tlogin where login = “or 1=1”. As 1=1 is evaluated as true, the entire where condition now becomes a tautology. The dynamic query now becomes “select id, level from tlogin where true”.

Server-side code injection attack detection based on KLD
Figure 1  Code example having SQLI vulnerability

```
1 $login = $_GET['Login'];
2 $pwd = $_GET['Password'];
3 $query = "select id from t1 where login ="\"$login.\" and password ="\"$pwd.\";"
4 $result = mysql_query ($query);
5 while($row = mysql_fetch_array ($result)) {
6     $_SESSION['ID'] = $user['id']; // authentication
7 }
```

The altered query retrieves all the record from the table. The attacker can easily skip the authentication mechanism by obtaining a valid session id.

2.2 An example of XSS attack

Figure 2(a) shows an example of HTML form that takes a user name and then allow a user to submit the name to the server side of the application (show.php script is invoked). The supplied input is accessed in `show.php` as shown in Figure 2(b). Here, `$_POST['user']` variable retrieves the supplied input and displays in the response page (`echo` statement). The displayed content is not filtered for JavaScript or HTML code elements.

```
<form action = "show.php" method= "post">
    Hello <?php echo $_POST['user']; ?>
</form>
```

If an attacker supplies JavaScript code, it would execute when the response page is being displayed in the victim’s browser. For example, if `name` field is supplied with `alert('xss')`, then browser displays a popup window with `xss` message. As the supplied code is directly reflected back in the response page, this type of XSS is called reflected attack. Another variation is called stored XSS attack where injected payload is stored in the database.

3 Related work

3.1 Related work on SQLI detection

In this section, we briefly discuss some related works that detect of SQLI attacks.

Agosta et al. (2012) identify SQLI vulnerabilities based on static taint analysis approach where inputs from users or files are marked as tainted, and their propagations are monitored into the relevant sinks (query invocation points). Buchrer et al. (2005)
Server-side code injection attack detection based on KLD

detect SQLI by comparing the parse tree of an intended SQL query before and after the inclusion of user supplied input. Boyd and Keromytis (2004) randomise SQL keywords so that attacker injected keywords become meaningless to the database engine. Halfond and Orso (2005) apply non-deterministic finite automata models of intended queries and identify invalid SQL queries due to injection attack which do not get consumed by the automata at runtime. Thomas et al. 2007 propose to convert SQL statements into prepared statements that do not alter during runtime. Kim (2011) detects SQLI attacks by first removing query attribute values or input variables that become part of dynamic queries and identifying the fixed values present in queries. At runtime, the attribute values are removed again from the generated queries and XOR operations are performed between the fixed values of the generated and earlier saved queries. An attack results in a non-zero value during the XOR operation.

Bandhakavi et al. (2007) develop the CANDID tool where they generate the expected parse trees of SQL queries with benign inputs (profiling). They compare them with the trees generated from actual inputs to detect any deviation and SQLI attacks. Lin and Chen (2007) propose application level security gateway to prevent SQLI attacks. They identify possible entry points of SQLI attacks to be protected by employing meta-programmes that can filter out meta-characters to avoid SQLI attacks. Kemalis and Tzouramanis (2008) express SQL queries with extended Backus Naur form (EBNF). They embed an architecture in programme code to monitor SQLI attacks by matching runtime generated queries with EBNF specifications. Interested readers may see the details of further works at Shahriar and Zulkernine (2012) based on testing, static analysis, dynamic analysis, and programme transformation.

The loginSystem-rc Project (2014) provides a collection of template pages for programmers to securely implement the login functionality to prevent SQLI attacks at the login interface of web applications. A number of open source scanner tools that can check SQLI vulnerabilities (W3af, 2013; Acutenix, 2013).

While all the above the efforts are useful at the implementation level, complementary runtime mitigation of SQLI attacks are needed. Our work can be applied in that direction.

### 3.2 Related work on XSS detection

XSS attack has been detected at the server-side. Shar and Tan (2012) develop a testing approach by extracting and modelling the implemented defences against XSS attacks as control flow graph, followed by checking the adequacy of defence mechanisms. Frenz and Yoon (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). They identify legitimate script code and compute a hash value which can then be compared with the hash of the actual script code present in a response page. Gundy and Chen (2009) apply HTML element namespace randomisation at the server side followed by derandomising 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

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 and Berghe, 2005; Futoransky et al., 2007; Chin and Wagner, 2009) to detect XSS attacks. The idea is to mark inputs as tainted, propagate taint information in programme operations, and check specific programme locations where tainted data is used. However, the analysis requires reimplemention of library APIs and interpreters so that programme operations become taint-aware.

Similar to all these above approaches, we develop a server-side framework to identify the legitimate script code and capture the distribution of script tokens based on JavaScript language. The probability distribution is then compared at runtime based on the distribution of script code present in an actual response page to identify the injected code presence using KLD.

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 and Doi (2009) prevent XSS attacks by first rendering the tree structure (DOM view) of a web page followed by binding parameters at trees 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.

We are aware of a number of scanner tools that can check XSS vulnerabilities (W3af, 2013; Acutenix, 2013). Our proposed approach can be considered as complementary to all the above approaches. Being our approach relying on KLD, the detection is anomaly-based. Hence, it brings the additional advantage of detecting unknown XSS attacks.

### 3.3 Related work on KLD

Several works apply the concept of KLD. Bigi (2003) applies KLD to identify authorship of documents. A model is built for each author by aggregating documents and developing a set of candidate models. Then, for a given document of unknown author, the approach finds the smallest KLD between a known candidate document and the document under test. Similar to this work, we apply constant back-off smoothing technique to address the missing elements (or tokens generated from injected JavaScript code or SQL keywords coming from attackers). We compare the KLD between expected structural elements and actual elements of a response page at the server-side.

Tapiador and Clark (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 SQLI and XSS attacks.
Li and Wang (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) measure 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 modularisation in non-object oriented software systems.

Khayam (2008) develop a worm detection technique by comparing the current traffic patterns (between source and destination ports) of a given host to the corresponding benign traffic patterns. The deviation between two traffic patterns is computed using Kullback-Leibler divergence. Lee and Xiang (2001) also apply KLD (along with other information-theoretic measures such as entropy, conditional entropy) to study the characteristics of audit log data and develop an anomaly detection model.

4 KLD-based detection of code injection attacks

In this section, we first briefly introduce the KLD measurement formula. Then, we discuss the server side framework for code injection attack detection. 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 SQLI and XSS attack detection using KLD with smoothing technique.

4.1 Kullback-Leibler divergence

The KLD computes the distance between two given probability distributions. Let us assume that \( P \) and \( Q \) represent two probability distributions of sets each having \( n \) number of similar elements with various probability occurrence. We denote \( P = \{p_1, \ldots, p_n\} \) and \( Q = \{q_1, \ldots, q_n\} \), where \( p_i \) is the occurrence of \( i^{th} \) element in \( P \) set, and \( q_i \) is the occurrence of \( i^{th} \) element in \( Q \) set. Then, the KLD is defined as follows (Cover and Thomas, 2006):

\[
\text{KLD}(P, Q) = \sum_{i} p_i \times \log_2 \left( \frac{p_i}{q_i} \right)
\]

Note that KLD is not symmetric (i.e., \( \text{KLD}(P, Q) \neq \text{KLD}(Q, P) \)). A challenge in computing \( \text{KLD}(P, Q) \) is the case where \( i^{th} \) element in either \( P \) or \( Q \) may be missing. In this case, \( p_i \) or \( q_i \) will be zero which would lead an infinite value while computing \( \log_2 \left( \frac{p_i}{q_i} \right) \). Thus, KLD would be infinite. To address this issue, we apply the constant back-off smoothing algorithm (Ney et al., 1994). Below, we briefly discuss the algorithm followed by example of attack detection.

4.1 Framework

Figure 3 shows a diagram of the proposed attack detection approach based on KLD. It has two stages: P set computation and instrumentation, and runtime detection.
During the first stage, the programme source is analysed to identify SQL query invocation points and JavaScript code segments. In particular, the static SQL query is parsed and tokenised to identify SQL keywords (e.g., SELECT, WHERE, FROM, OR, AND, and LIKE) and their occurrence probability is obtained to form the P set for SQLI attack detection. Similarly, the JavaScript code segments are gathered and parsed to compute the P set for XSS attack detection. Here, we identify the occurrence probability of built-in objects (document, window), and method calls (e.g., open, alert, eval) for XSS attack detection. Currently, we consider all inline JavaScript code (e.g., `<script> ... </script>`, code present as URL attribute (e.g., `<img src= “javascript:...”>`) and CSS (e.g., `<div style = “background:url (‘javascript:... ’)”`). If the JavaScript code is encoded, then it is converted to plain text. The obtained P sets are stored in a repository for comparison at a later time with Q set.

**Figure 3** KLD-based code injection detection framework

![Diagram](https://example.com/diagram.png)

Based on the P set, the source code is instrumented to check the KLD based on the obtained Q set at runtime. Figure 4 shows an example of code instrumentation for detecting SQLI attack (lines 4 and 5 are added for the checking based on KLD value exceeding a threshold t). Here, we assume that the static query structure is denoted as q1, the runtime query is denoted as $query$, the threshold is denoted as t, and $KLD\_Compute()$ method measures the distance between probability distribution of tokens obtained from the static query (q1) and the dynamic query ($query$). If the deviation exceeds a threshold level t, then the response page is redirected to the login page again (login.php) so that attacker is not able to bypass authentication or obtain confidential information.
Figure 4  Code instruction example for checking the occurrence of SQLI attack using KLD measure

```php
1 $login = $_GET['Login'];
2 $pwd = $_GET['Password'];
3 $query = "select id from t1 where login ="'. $login. " and password ="'. $pwd. "";
4 if ( KLD_Compute ('q1', $query) > t)
5  redirect('login.php');
6 $result = mysql_query ($query);
7 while($row = mysql_fetch_array ($result)) {
8  $_SESSION['ID'] = $user['id']; // authentication
8 }
```

During the second stage (the bottom of Figure 3), we compute the $Q$ set for respective attack types (i.e., parse SQLI query or JavaScript code present in present in response page). The $P$ and $Q$ sets are smoothed (more discussion on this step is provided in Section 4.3) due to missing tokens in any of the two sets, and we obtain revised sets denoted as $P'$ and $Q'$. The KLD between $P'$ and $Q'$ are computed and matched against a threshold level ($t$) to detect code injection attacks.

4.3 Computing KLD with smoothed probability distributions

The key idea of smoothing probability distribution is to add a very negligible number $e$ for missing elements probability occurrence, and then subtracting the probability of other elements proportionately, so that the sum of the probability of all elements become one. Note, after we smooth $P$ and $Q$, we denote them as $P'$ and $Q'$, respectively. Figure 5 shows the algorithm of back-off smoothing that accepts $P$, $Q$, and $e$, and returns the KLD of the two smoothed sets $P'$ 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 (line 5). $CU$ 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 line 8. It can be shown that $pc = e^{|SU-SP|/|P|}$. Line 9 applies smoothing for the $Q$ set and generates $Q'$, and it can be shown that $qc = e^{|SU-SP|/|Q|}$. The KLD value is then computed for $P'$ and $Q'$ in line 10.
Figure 5  Algorithm for computing KLD based on smoothing

Algorithm KLD_Backoff_Smoothing \((P, Q, e)\)

1. \(P = \{p_i\}, \text{ where } p_i \in P\).
2. \(CP = |P|\).
3. \(Q = \{q_i\}, \text{ where } q_i \in Q\).
4. \(CQ = |Q|\).
5. \(SU = P \cup Q\).
6. \(CU = |SU|\).
7. \(P'\) contains all elements from \(SU\), and the probability of each element \(p'\) is calculated as follows:
   \[ p'_i = p_i - pc \text{ if } i^{th} \text{ element is in } P \]
   \[ p'_i = e, \text{ otherwise} \]
8. Compute \(pc\) by solving \(\sum p_i = 1.0\).
9. Similarly smooth \(Q\) to generated \(Q'\) and identify \(qc\) by solving \(\sum p_i = 1.0\).
10. \(d = \sum p_i \times \log_2 (p'_i / q'_i)\).
11. return \(d\).

4.4 An example of SQLI attack detection based on KLD

We reuse the earlier example of SQL query from Figure 1 to illustrate the application of KLD measure to detect SQLI attacks. Here, the static query “select id from t1 where login = ‘$login’ and password = ‘$pwd’” would generate the following \(P\) set (SQL keyword occurrence probability).

\[ P = \{\text{select:}1/4, \text{from:}1/4, \text{where:}1/4, \text{and:}1/4\} \]

The dynamic query having tautology attack input (select id, level from tlogin where login = ‘’ or l=1 -- and password = ‘’$pwd’’); would generate the following \(Q\) set as follows:

\[ Q = \{\text{select:}1/5, \text{from:}1/5, \text{where:}1/5, \text{or:}1/5, \text{and:}1/5\} \]

We apply the algorithm of Figure 5 to generate smoothed population \(P'\) and \(Q'\) from \(P\) and \(Q\), respectively as follows:

\[ CP = 4, CQ = 5, SU = \{\text{select, from, where, or, and}\} \]
\[ P' = \{\text{select:}1/4 - e, \text{from:}1/4 - e, \text{where:}1/4 - e, \text{or:}4e, \text{and:}1/4 - e\} \]
\[ Q' = \{\text{select:}1/5, \text{from:}1/5, \text{where:}1/5, \text{or:}1/5, \text{and:}1/5\} \]

We now compute KLD between \(P'\) and \(Q'\) in Table 1 assume \(e = 0.00001\). The value generated in the last column is summed up to find the total KLD which is 0.321. If we assume that the threshold value to compare this KLD is a very small number (e.g., 0.001), then an attack will be detected. We discuss some choices of possible threshold values in Section 5.
Table 1: Detailed computation of KLD between \( P' \) and \( Q' \) for SQLI attack detection

<table>
<thead>
<tr>
<th>Element</th>
<th>( p'_i )</th>
<th>( q'_i )</th>
<th>( \log(p'_i) )</th>
<th>( \log(q'_i) )</th>
<th>( p'_i \ast (\log(q'_i) - \log(q'_i)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>0.24999</td>
<td>0.2</td>
<td>-2.00006</td>
<td>-2.32193</td>
<td>0.080464378</td>
</tr>
<tr>
<td>from</td>
<td>0.24999</td>
<td>0.2</td>
<td>-2.00006</td>
<td>-2.32193</td>
<td>0.080464378</td>
</tr>
<tr>
<td>where</td>
<td>0.24999</td>
<td>0.2</td>
<td>-2.00006</td>
<td>-2.32193</td>
<td>0.080464378</td>
</tr>
<tr>
<td>or</td>
<td>0.00004</td>
<td>0.2</td>
<td>-14.6096</td>
<td>-2.32193</td>
<td>-0.000491508</td>
</tr>
<tr>
<td>and</td>
<td>0.24999</td>
<td>0.2</td>
<td>-2.00006</td>
<td>-2.32193</td>
<td>0.080464378</td>
</tr>
</tbody>
</table>

KLD \((P', Q')\) 0.321366003

4.5 An example of XSS attack detection based on KLD

We reuse the PHP code example shown in Figure 2(b) where we substitute the PHP variable \( \_POST['user'] \) with an injected JavaScript code \(<script>alert(document.cookie);</script>\).

We apply the algorithm of Figure 5 to generate smoothed population \( P' \) and \( Q' \) from \( P \) and \( Q \), respectively as follows:

\[
CP = 1, \ CQ = 4; \ SU = \{s1, script, alert, document, cookie\}
\]

\[
P' = \{s1:1-4e, script:e, alert:e, document:e, cookie:e\}
\]

\[
Q' = \{s1:4e, script:2/5-e, alert:1/5-e, document:1/5-e, cookie:1/5-e\}
\]

Table 3 shows the detailed computation of KLD between \( P' \) and \( Q' \). The last column shows the KLD value which is 14.608. If we assume that the threshold value to compare this KLD is a very low number such as 0.001, then an XSS 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 FP warnings for benign web pages.
Table 3 Detailed computation of KLD between $P'$ and $Q'$ for XSS attack detection

<table>
<thead>
<tr>
<th>Element</th>
<th>$p'_i$</th>
<th>$q'_i$</th>
<th>$\log(p'_i)$</th>
<th>$\log(q'_i)$</th>
<th>$p'_i \cdot \left(\log(q'_i) - \log(q'_i)\right)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>0.99996</td>
<td>0.00004</td>
<td>-5.7709E-05</td>
<td>-14.60964047</td>
<td>14.60899838</td>
</tr>
<tr>
<td>script</td>
<td>0.00001</td>
<td>0.39999</td>
<td>-16.60964047</td>
<td>-1.321964163</td>
<td>-0.000152877</td>
</tr>
<tr>
<td>alert</td>
<td>0.00001</td>
<td>0.19999</td>
<td>-16.60964047</td>
<td>-2.322000231</td>
<td>-0.000142876</td>
</tr>
<tr>
<td>document</td>
<td>0.00001</td>
<td>0.19999</td>
<td>-16.60964047</td>
<td>-2.322000231</td>
<td>-0.000142876</td>
</tr>
<tr>
<td>cookie</td>
<td>0.00001</td>
<td>0.19999</td>
<td>-16.60964047</td>
<td>-2.322000231</td>
<td>-0.000142876</td>
</tr>
<tr>
<td>KLD ($P', Q'$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14.60841688</td>
</tr>
</tbody>
</table>

5 Experimental evaluation

We evaluate the proposed approach with three PHP applications (obtained from sourceforge.net). These applications have been reported to contain SQLI and XSS vulnerabilities. These applications include **PHP-Address book** (address and contact manager; PHP Address Book, 2014), **Serendipity** (blog management system; Serendipity 2014), and **PHP-fusions** (content management system; PHP-Fusion, 2014). These applications rely on MySQL databases to store and retrieve information.

Table 4 shows the characteristics of these applications that include the number of PHP files we analyse (column 2), the number of SELECT, INSERT, UPDATE, and DELETE queries instrumented for checking SQLI attacks (columns 3 to 6), the number of `echo` statements (column 7) instrumented for checking XSS attacks, and the three types of JavaScript code (inline, URL attribute, and CSS) (columns 8 to 10).

Table 4 Characteristics of PHP applications

<table>
<thead>
<tr>
<th>Application</th>
<th># of files</th>
<th>SELECT</th>
<th>INSERT</th>
<th>UPDATE</th>
<th>DELETE</th>
<th><code>echo</code></th>
<th>Inline</th>
<th>URL</th>
<th>CSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>312</td>
<td>25</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>567</td>
<td>7</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>Serendipity</td>
<td>1,344</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1,619</td>
<td>56</td>
<td>43</td>
<td>126</td>
</tr>
<tr>
<td>PHP-fusions</td>
<td>718</td>
<td>62</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>4,555</td>
<td>80</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

5.1 Attack input selection

To evaluate SQLI attack detection, we choose four attack types (Halfond et al., 2006): tautology ($sql_0$), union query ($sql_1$), piggy-backed query ($sql_2$), and inference attack ($sql_3$) (applicable for MySQL database). Table 5 shows the examples of attack inputs.

Table 5 SQLI attack types

<table>
<thead>
<tr>
<th>Attack types</th>
<th>Example attack input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tautology ($sql_0$)</td>
<td>' or 1=1 --</td>
</tr>
<tr>
<td>Union ($sql_1$)</td>
<td>' UNION SELECT 1,1 --</td>
</tr>
<tr>
<td>Piggy backed query ($sql_2$)</td>
<td>' ; show tables; --</td>
</tr>
<tr>
<td>Inference ($sql_3$)</td>
<td>' ;/*! select concat(‘1’,’2’) */g --</td>
</tr>
</tbody>
</table>
To evaluate XSS attacks, we randomly choose five types of malicious script code from OWASP-XSS-Cheat-Sheet (2014). Table 6 shows the description of each type of XSS attack along with example payload \((xss_0-a_4)\). The attacks include injecting JavaScript through HTML attributes and invoking `onload` \((xss_0)\), `onmouseover` \((xss_1)\), `onerror` \((xss_2)\) method calls, encoded URI \((xss_3)\), and code encoding \((xss_4)\).

### Table 6  XSS attack types

<table>
<thead>
<tr>
<th>Description</th>
<th>Attack payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS in Attributes – onload ((xss_0))</td>
<td><code>&lt;body onload=alert('test1')&gt;</code></td>
</tr>
<tr>
<td>XSS in Attributes – onmouseover ((xss_1))</td>
<td><code>&lt;b onmouseover = alert('Wufff!')&gt; click me!&lt;/b&gt;</code></td>
</tr>
<tr>
<td>XSS in Attributes – onerror ((xss_2))</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 ((xss_3))</td>
<td><code>&lt;IMG SRC=j&amp;amp;#X41script&gt;alert('test2')&gt;</code></td>
</tr>
<tr>
<td>XSS using code encoding ((xss_4))</td>
<td><code>&lt;META HTTP-EQUIV=&quot;refresh&quot; CONTENT=&quot;0;url=data:text/html;base64,PHNjcmlwbdD5hbGVydCgdGVzdlDMnKTwvc2NyaXB0Pg&quot;&gt;</code></td>
</tr>
</tbody>
</table>

### Table 7  List of SQLI and XSS vulnerabilities from OSVDB

<table>
<thead>
<tr>
<th>Application</th>
<th>OSVDB ID (SQLI)</th>
<th>OSVDB ID (XSS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>92,534, 92,535, 92,106, 92,105, 92,104, 92,103, 92,102, 92,101, 92,100, 92,099, 92,098, 92,097, 92,096, 80,832, 80,834, 82,264, 45,965, 46,091, 44,425</td>
<td>92,537, 81,984, 81,986, 81,987, 81,985, 80,833, 45,966, 53,389, 43,668</td>
</tr>
<tr>
<td>Serendipity</td>
<td>82,036, 81,773, 58,111, 34,935, 15,875, 15,542, 10,370, 10,371</td>
<td>95,176, 87,395, 81,713, 76,836, 76,856, 75,777, 75,484, 72,726, 67,693, 68,944, 26,999</td>
</tr>
<tr>
<td>PHP-fusions</td>
<td>90,714, 90,713, 90,712, 90,711, 90,710, 90,709, 90,695, 90,693, 90,359, 70,839, 70,451, 68,621, 57,913, 52,542, 51,997, 51,998, 51,440, 50,992, 50,065, 48,822, 49,041, 49,878, 49,041, 44,532, 38,593, 37,410, 37,411, 36,310</td>
<td>90,697, 90,708, 90,707, 90,706, 90,705, 90,704, 90,703, 90,702, 90,701, 90,700, 90,699, 90,698, 90,680, 87,961, 75,485, 51,053, 36,342</td>
</tr>
</tbody>
</table>

### 5.2 Setup and test case selection

To identify the locations where attack inputs can be applied, we rely on the reported vulnerabilities from OSVDB\(^2\) 2014. Table 7 shows the list of OSVDB ID (SQLI and XSS) that we examine to identify the responsible pages that can be used to inject SQL injection attack or arbitrary JavaScript code. We identify the URLs for injecting SQLI attack inputs. If multiple parameters are applicable for injecting SQLI inputs, we randomly choose one parameter. We apply XSS attack inputs for all the relevant pages that allow viewing/rendering the injected JavaScript code. In both of the attack types, we examine whether the injected code is detected or not by comparing the KLD values with
the threshold level. Note during this evaluation, the constant probability \((e)\) used for the back-off smoothing algorithm is 0.0001, and KLD threshold level is set as 0.001.

5.3 Result summary for SQLI attack detection

Figures 6, 7, and 8 show the KLD values obtained while evaluating the three applications as we inject each of the four types of SQLI attacks: tautology (\(sqli_0\)), union (\(sqli_1\)), piggyback (\(sqli_2\)), inference (\(sqli_3\)). Table 8 shows a high level summary of the evaluation results. We show the minimum and maximum level of KLD obtained for all the attacks in columns 2 and 3, respectively. The last two columns show the total number of attack inputs injected and the total number of attack inputs detected. We observe that with a threshold value of 0.001, KLD-based approach is able to detect all the injected SQLI attack inputs.

**Figure 6** KLD values of PHP Address book application for testing \(sqli_0–sqli_3\) (see online version for colours)

**Figure 7** KLD values of Serendipity application for testing \(sqli_0–sqli_3\) (see online version for colours)
Server-side code injection attack detection based on KLD

Figure 8  KLD values of PHP-fusions application for testing $sql_inj-sqli_3$ (see online version for colours)

Table 8  Evaluation summary for SQLI attack detection

<table>
<thead>
<tr>
<th>Application</th>
<th>KLD (min)</th>
<th>KLD (max)</th>
<th>Total attack inputs applied</th>
<th>Total attack input detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>0.12</td>
<td>8.95</td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>Serendipity</td>
<td>0.19</td>
<td>5.26</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>PHP-fusions</td>
<td>0.02</td>
<td>6.64</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>

We then vary the threshold level ($t$) for comparing KLD to 0.05, 0.5, and 1.0 and observe the number of undetected SQLI attacks. A summary is shown in Table 9. The last row computes the false negative (FN) rate (ration between the total number of undetected attacks and the total number of injected attacks). We notice that as the value of $t$ increases, FN rate increases. Thus, we suggest choosing a very low number for threshold value to effectively detect SQLI attacks.

Table 9  SQLI detection effectiveness for various threshold levels

<table>
<thead>
<tr>
<th>Application</th>
<th>t = 0.05</th>
<th>t = 0.5</th>
<th>t = 1.0</th>
<th>Total # of attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>0</td>
<td>11</td>
<td>25</td>
<td>76</td>
</tr>
<tr>
<td>Serendipity</td>
<td>0</td>
<td>2</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>PHP fusions</td>
<td>2</td>
<td>11</td>
<td>30</td>
<td>128</td>
</tr>
<tr>
<td># of undetected attack (FN%)</td>
<td>2 (0.8%)</td>
<td>24 (10.1%)</td>
<td>75 (31.7%)</td>
<td>236</td>
</tr>
</tbody>
</table>

Table 10  Evaluation summary of FP (FP%) for various threshold levels

<table>
<thead>
<tr>
<th>Application</th>
<th>Total # of benign inputs</th>
<th>t = $7.19*10^{-7}$ (FP%)</th>
<th>t = $5.19*10^{-2}$ (FP%)</th>
<th>t = $0.002$ (FP%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>108</td>
<td>0 (0%)</td>
<td>1 (0.9%)</td>
<td>1 (0.9%)</td>
</tr>
<tr>
<td>Serendipity</td>
<td>132</td>
<td>1 (0.7%)</td>
<td>2 (1.5%)</td>
<td>3 (2.2%)</td>
</tr>
<tr>
<td>PHP fusions</td>
<td>204</td>
<td>2 (0.9%)</td>
<td>3 (2.4%)</td>
<td>5 (2.4%)</td>
</tr>
<tr>
<td># of false alarm (FP%)</td>
<td>444</td>
<td>3 (0.6%)</td>
<td>6 (1.2%)</td>
<td>9 (2.0%)</td>
</tr>
</tbody>
</table>
We then evaluate the FP rate by supplying randomly generated strings (drawn randomly from online new sources) into HTML form fields, and then viewing the KLD values for possible false attack warning. Table 10 shows a summary of the FP evaluation for three low threshold values chosen randomly ($t = 7.19 \times 10^{-7}$, $t = 5.19 \times 10^{-2}$, $t = 0.002$) for the three applications. The last row shows the overall number of benign inputs, total number of false warning along with FP rate.

We apply total 108, 132, and 204 benign inputs for *PHP Address book*, *Serendipity*, and *PHP fusions*, respectively. The FP rate ranges between 0.0% and 0.9%, 0.7% and 2.2%, and 0.9% and 2.4% for *PHP Address book*, *Serendipity*, and *PHP fusions*, respectively. The overall FP rate is between 0.6% and 2.0%. Our observation indicates that setting the threshold value ($t$) close to zero results in avoiding FP warnings. The FP rates are due to inputs having successive meta characters (e.g., `, ~) and matching of SQL like operators (AND, OR) that are also used for legitimate purpose in the real-world in words or sentences.

### 5.4 Result summary for XSS attack detection

Figures 9, 10, and 11 show the KLD values obtained while evaluating XSS attack detection effectiveness for the five attack types ($\text{xss}_i$–$\text{xss}_5$).

Table 11 shows a high level summary of the computed KLD during our evaluation of XSS attack detection effectiveness. The second and third columns show the minimum and maximum KLD values obtained for each of the applications. 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, 15.72). The last two columns show the total number of attack inputs applied and the total number of detected attacks. Considering the threshold value of 0.001, KLD-based approach is able to detect all the injected XSS attacks.
We now vary the KLD threshold level \( t \) to 0.05, 0.5, and 1.0 and observe the number of undetected XSS attacks. A summary is shown in Table 12. The last row computes the FN rate. We notice that as the value of \( t \) increases, FN rate increases for XSS attack detection (similar to the observation for SQLI attack detection). Thus, we suggest choosing a very threshold value to detect XSS attacks using KLD-based approach.

**Figure 10**  KLD values of serendipity application for testing XSS\(_{0}\)–XSS\(_{3}\) (see online version for colours)

**Figure 11**  KLD values of PHP-fusions application for testing SQLI\(_{0}\)–SQLI\(_{3}\) (see online version for colours)

**Table 11**  Evaluation summary for XSS attack detection

<table>
<thead>
<tr>
<th>Application</th>
<th>KLD (min)</th>
<th>KLD (max)</th>
<th>Total attack inputs applied</th>
<th>Total attack input detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHP-Address book</td>
<td>0.35</td>
<td>15.55</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Serendipity</td>
<td>0.02</td>
<td>15.93</td>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>PHP-fusions</td>
<td>0.03</td>
<td>15.72</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>
Table 12  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 FP rate by supplying randomly generated benign strings into HTML form fields, and then observing if KLD value exceeds the chosen threshold for false warning. Table 13 shows a summary of the FP evaluation for three randomly chosen 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 total number of benign inputs applied, the total number of false warning along with FP rate (ratio between the number of false warning to the total number of benign inputs).

Table 13  Evaluation summary of 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 benign inputs depend on input form fields present in a test page such as random texts, numbers, and dates. 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%, \(t = 5.19 \times 10^{-7}\)). The overall FP rate is between 2.0% and 3.6%. Similar to SQLI, our observation indicates that setting the threshold value \(t\) close to zero results in avoiding FP warnings.

5.5 Overhead evaluation

We assess the overhead imposed by the KLD-based detection approach. We measure the time delay required to receive a response page at the client side after supplying inputs with benign or malicious inputs. Table 14 shows the average delay to receive a response page during our evaluation. We observe that PHP Fusions has the highest average delay (567 ms), followed by the PHP Address Book (543 ms) and Serendipity (332 ms). The major reason of having the delay is to extract and tokenise SQL queries and JavaScript code for computing \(Q\) set and applying the smoothing algorithms to compute the KLD. Some the response pages that have large amount of inline JavaScript code and large SQL queries (nested) that require more time to compute KLD.
6 Conclusions and future work

SQLI and XSS are the two most common code injection vulnerabilities discovered widely in web applications. These vulnerabilities can lead to many unwanted security breaches such as authentication bypassing and deletion of information. To mitigate the code injection attacks, this paper proposes a server-side attack detection approach based on KLD measure. Our approach relies on distance between the probability distribution of legitimate and the observed structure (e.g., SQL query, JavaScript code) present in a response page. The high deviation between the two structures indicates an attack. We also address the issue of computational challenge of infinite KLD by applying the constant back-off smoothing technique. We evaluate the approach with three vulnerable PHP applications. The evaluation results indicate that the approach can successfully detect most of the known SQLI and XSS attacks based on appropriate KLD threshold values. The estimated FN rates vary between 0% and 31%, and the FP rates vary between 0.6% and 3.6%. We also show a comparative analysis for designer to choose the most appropriate threshold values to minimise FP and FN rates. The approach imposes negligible overhead.

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. Currently, our approach does not address SQLI detection for stored procedures, and illogical queries. We also plan to evaluate the approach for these SQLI attacks.

References


Notes
1 An earlier version of this article has been published in Shahriar et al. (2013).
2 We considered vulnerability reported between the years 2007–2013.