

EARLY DETECTION OF SQL INJECTION ATTACKS

Hossain Shahriar¹, Sarah North², and Wei-Chuen Chen³

Department of Computer Engineering, Kennesaw State University, Georgia, USA

¹hshahria@kennesaw.edu

²snorth@kennesaw.edu

³wchen10@students.kennesaw.edu

ABSTRACT

SQL Injection (SQLI) is a common vulnerability found in web applications. The starting point of SQLI attack is the client-side (browser). If attack inputs can be detected early at the browse side, then it could be thwarted early by not forwarding the malicious inputs to the server-side for further processing. This paper presents a client-side approach to detect SQLI attacks¹. The client-side accepts shadow SQL queries from the server-side and checks any deviation between shadow queries with dynamic queries generated with user supplied inputs. We measure the deviation of shadow query and dynamic query based on conditional entropy metrics and propose four metrics in this direction. We evaluate the approach with three PHP applications containing SQLI vulnerabilities. The evaluation results indicate that our approach can detect well-known SQLI attacks early at the client-side and impose negligible overhead.

KEYWORDS

SQL Injection, Web security, Conditional entropy, Information theory

1. INTRODUCTION

SQL Injection (SQLI) vulnerability is a well-known security concern for web applications that alters the implemented query structures with supplied malicious inputs. The execution of altered queries may lead to security breaches such as unauthorized access to application resources, escalation of privileges, and modification of sensitive data [33]. A number of recent surveys [1, 12] indicate that SQLI is among top three worst vulnerabilities discovered in today's web-based applications after their deployment. Moreover, a large portion of data security breaches have been caused by SQLI attacks in real world resulting in financial losses to business organizations [8]. So, detecting SQLI attacks early can reduce the potential losses.

The starting point of SQLI attack is the client-side (browser). If attack inputs can be detected early at the browse side, then it could be thwarted early by not forwarding the malicious inputs to the server-side for further processing. This could bring two benefits: adding an extra protection layer on top of server-side solutions (e.g., secure coding [9, 31], code generation [33], dynamic analysis [4, 29]) and working as a complementary approach to other known existing black-box level solutions (e.g., security scanner tools [14, 15, 19, 28, 32]). However, there are challenges to develop a client-side SQLI attack detection approach. For example, the relevant form fields that require checking at the client-side must be informed by the server-side and the information should not contain sensitive data such as original query, table name and column name. The deviation of dynamic query structure at the client-side needs to be computed and checked based on the information supplied from the server-side. So, suitable measurements are required at the client-

¹ An earlier version of this work has been published in [34].

side. Finally, the approach should be light-weight which means heavy computation should not be performed at the client-side.

This paper attempts to address these issues. We propose a client-side SQLI attack detection framework based on conditional entropy metrics. We address the first issue by developing an approach to extract query structures from the server-side code and convert them to shadow queries. A shadow query is identical to an original query, except column names, table names, and input variables are replaced with arbitrary symbolic values. The second issue is addressed by measuring the deviation of information content between shadow queries with symbolic values and actual input values. We leverage the concept of information theory [25, 26] to measure the information content of queries. In particular, our contribution remains in the development of four types of conditional entropy metrics. If there is a deviation of the information content between a shadow query and a dynamic query (formed by replacing symbolic values with actual inputs from form fields), then an SQLI attack is detected at the client-side and a request is blocked. Otherwise, inputs supplied to a form are considered as benign and a request is allowed to the server-side.

We evaluate our approach with three PHP web applications containing known SQLI vulnerabilities. The evaluation results indicate that the approach can detect most common attack inputs at the client-side. Moreover, the approach has negligible overhead in terms of computation and code instrumentation. The proposed approach has several advantages. First, it does not rely on the specific type of attack inputs and new attack inputs can be detected. Second, input filtering mechanisms implemented at the client and server-sides may not detect all malicious inputs, so it can be a complementary approach to other existing solutions [14]. Further, the approach imposes negligible overhead at the browser.

The paper is organized as follows. Section 2 shows an example of SQLI attack at the client-side followed by a brief introduction of our approach for detecting it. We also discuss the related work. In Section 3, the proposed client-side SQLI attack detection framework is discussed in details followed by conditional entropy metrics and their application in SQLI attack detection process. Section 4 describes the experimental results found during our evaluation. Section 5 concludes the paper and discusses future work.

2. BACKGROUND AND RELATED WORK

In this section, we first provide an example of SQL Injection attack occurrence followed by a brief overview how our approach can detect the attack. Then, we discuss some related work on SQL Injection from the literature.

2.1. Example of SQL Injection attack and detection by our approach

First, we explain how an SQLI attack can occur in PHP code. We show a login form in Figure 1(a). The form has two input fields (*Login*, *Password*) whose values are supplied by a user. When a user clicks on *Login* button, a request for accessing the website is sent to the server side with the supplied values. Figure 1(b) shows the HTML code of the HTML form. Here, the *Login* and *Password* input fields are accessed at the server-side through *\$fLogin* and *\$fPassword* variables, respectively. Also, the *login.php* script is executed at the server-side which generates a dynamic SQL query for authentication based on the supplied values.

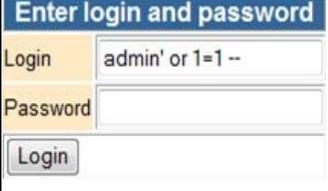
(a) Login form	(b) HTML of login form
	<pre data-bbox="630 254 1326 445"><form name = "form1" action="login.php" method="post"> <input type="text" name= "fLogin"> <input type= "text" name= "fPassword"> <input type= "submit" name= "Login"> </form></pre>

Figure 1. (a) A login form. (b) HTML code of login form.

Figure 2 shows the PHP code snippet. Lines 1 and 2 extract the *fLogin* and *fPassword* fields of a request into *\$login* and *\$pwd* variables, respectively. Note that the inputs are not filtered and directly concatenated with other strings for generating a dynamic SQL query at Line 3. Thus, the code is vulnerable to SQLI attack. Line 4 executes the query and Line 5 performs the authentication based on the presence of at least one row in the result set. If the provided credential information is matched in the *tlogin* table, the current session ID is set with the obtained *id* field from the table.

```

1. $login = $_POST['fLogin'];
2. $pwd= $_POST['fPassword'];
3. $qry = "select id, level from tlogin where uid ='" . $login. "' and password ='" . $pwd. "'";
4. $result = mysql_query($qry);
5. while($row = mysql_fetch_array($result)) { // authentication
6.     $_SESSION['ID'] = $row['id'];
.....
7. }
```

Figure 2. PHP code for authentication.

If a user provides benign values for the input fields as “*admin*” and “*secret*” respectively, the dynamic query at Line 3 becomes as follows: *select id, level from tlogin where uid= 'admin' and password = 'secret'*. However, if a user supply malicious inputs as shown in Figure 1(a), the query becomes: *select id, level from tlogin where uid= '' or 1=1 --' and password = ''*. The resultant query is now a tautology. Here, the query part after the comment symbol “*--*” is ignored. The remaining condition *uid= '' or 1=1* is evaluated as *true*. Therefore, the supplied malicious input has altered the intended query structure.

We now show how our approach can detect the attack from the client-side. We modify the HTML form as shown in Figure 3 (a) by adding three hidden fields. They represent server-side provided information on shadow query (*sQuery*), the form fields that need to be checked (*chkField*), and the values that need to be substituted with the form field values (*substitute*) in a shadow query. In addition, a JavaScript code is supplied from the server-side to perform the checking of SQLI attack at the client-side. The method *chkSQLI* is shown briefly in Figure 3(b). Here, the script code extracts the shadow query at Line 2. Line 3 computes the conditional entropy of the shadow query (*entropySqry*). Line 4 substitutes shadow query’s symbolic values (specified at *document.form1.substitute* as comma separated field values) with form fields (specified at *document.form1.chkField* as comma separated values).

The conditional entropy (*entropyAqry*) of the actual query is obtained at Line 5. Line 6 compares the value of *entropyAqry* and *entropySqry*. If there is a match, the inputs are considered as benign and the form is submitted to the remote website (Lines 6-7). If there is a mismatch, a warning

message is generated on SQLI attack related inputs and the form is not submitted to the remote website (Lines 8-9).

<p>(a) Revised HTML form</p> <pre> <form name = "form1" action="login.php" method="post"> <input type="text" name= "fLogin"> <input type= "text" name= "fPassword"> <input type= "hidden" name= "sQuery" value= "select c1, c2 from t1 where c3 =v1 Ilc4 = v2"> <input type= "hidden" name= "chkField" value ="fLogin, fPassword"> <input type= "hidden" name= "substitutue" value ="v1, v2"> <input type= "submit" name= "Login" onclick= "chkSQLI ()"> </form> </pre>
<p>(b) JavaScript code for checking SQLI attack</p> <pre> 0. <script type="text/javascript"> 1. function chkSQLI(){ 2. var sqry = document.form1.sQuery; 3. var entropySqry = condEntropy(sqry); 4. var aqry = subs (sqry, document.form1.chkField, document.form1.substitutue); 5. var entropyAqry = condEntropy(aqry); 6. if (entropySqry == entropyAqry) 7. document.form1.submit(); 8. else 9. alert ("SQL injection attack inputs are present in supplied values"); 10. } 11. </script> </pre>

Figure 3. (a) Modified HTML form (b) JavaScript code for checking SQLI attack at client-side. We will discuss more about the framework, server-side shadow query generation process, client-side conditional entropy computation technique in Section 3.

2.2. Related work

Many works have addressed the detection of SQLI attacks from the server-side. The work of Kim *et al.* [30] is a recent work in this direction. They remove input variables from queries and identify the fixed values present in queries. At runtime, they remove attribute values from generated queries and perform XOR operation with the earlier saved fixed values. An attack results in a non-zero value during the XOR operation between. In contrast, our approach detects the attack at the client-side and relies on conditional entropy based comparison to identify malicious attack inputs.

SQLI attacks have been detected by comparing the parse tree of an intended query and altered query [2] and randomizing SQL keywords [3]. A number of approaches rely on taint-based static analysis [4, 29] which are often dependent on string analysis algorithms and suffer from runtime overhead and false positive warning at the server-side. In contrast to all these approaches, we put the burden of detecting SQLI attacks at the client-side with a priori information sent by the server-side. Other popular approach includes applying prepared statement which is language dependent and not generic [9] as well as profiling the parse trees generated by benign inputs [10].

Lin *et al.* [11] propose application level security gateway to prevent SQLI attacks. They identify possible entry points of SQLI attacks to be protected by employing meta-programs that can filter

out meta-characters to avoid SQLI attacks. Kemalset *al.* [13] express SQL queries with Extended Backus Naur Form (EBNF). They embed an architecture in program code to monitor SQLI attacks by matching runtime generated queries with EBNF specifications.

In one of our earlier works [24], we compute the entropy of SQL queries to detect SQLI attacks at the server side. This paper is different from our previous work as we now send information from the server side to the client side to detect malicious SQLI attack inputs based on four types of conditional entropies. In contrast, our previous work just relies on the single entropy measure of a given query to detect SQLI attacks at the server side without sending any information at the client side. The previous work may suffer from the attack detection effectiveness for input cases where the entropy level of expected queries and malicious queries might be similar. We alleviate this issue in this paper by proposing four different metrics to compare the deviation between expected and actual query to detect a wide range of malicious inputs.

Several works have addressed SQLI attacks from the perspective of effective test case generation at the server-side ([5, 7, 17]). Interested readers may see the details of further works in the survey of [21] based on testing, static analysis, dynamic analysis, and program transformation. In contrast to all these efforts, our proposed approach detects the attack at the client-side and provides the advantage of stopping the attacks early and avoiding consuming server-side resources during attack detection.

Our work is also motivated by a number of works based on information theory [26] as a measurement technique to address a variety of security problems such as worm containment [23], characterizing audit log data [16], and performance comparison of IDS [20]. Several works apply information theory to tackle non-security related issues such as measuring the complexity of software [18] and allocating resources of data storage systems [22]. In contrast to these work, we propose a set of conditional entropy metrics to detect SQLI attacks at the client-side. The idea of conditional entropy comes from the motivation that each of the query type dominates the participation of dynamic variables whose values are supplied from a user. So, instead of measuring the entropy of the whole query we can measure the entropy only part of a query in its symbolic representation form.

3. SQLI ATTACK DETECTION APPROACH

Figure 4 shows the proposed framework for detecting SQLI attacks at the client-side with priori information provided by the server-side. We briefly discuss the six steps (shown in Figure 4) to provide the workflow of the framework.

First, server-side script code is pre-processed to identify HTML forms that contain input fields. Then the SQL queries present in the script code is extracted and we find the relevant set of input fields in forms that contribute values during dynamic query generation process. We perform static backward slicing on PHP code to find the relevant variables and their reachability to form fields [27, 35]. The SQL query present at the server-side script is simplified to generate shadow query. This is done by replacing table, column, logical operator, and variable values with symbolic values. This step prevents revealing sensitive information at the client-side. The HTML forms are modified by adding hidden fields with information on the following: (i) the shadow query, (ii) fields relevant to dynamic queries and whose values are needed to be replaced with appropriate symbols in the shadow query, and (iii) the relevant symbols from shadow query that need to be replaced with user provided inputs. The step also includes the computation of conditional entropy of a shadow query and an actual query based on the query types and comparing the values for any deviation to detect SQLI attacks.

In the second step, the modified server-side script code is deployed in a web server. Then, from a browser, a user requests a web page to the server (third step) and it returns the modified pages containing revised HTML forms (fourth step). The user provides inputs in the form and resubmits to the server (step 5).

The client-side checking takes place in the JavaScript code during step 6. The checking is done by computing the conditional entropy of the shadow query and the actual query by substituting symbols with user provided input values. If any deviation is found, the inputs are flagged as malicious and a user is warned of attempting SQLI attacks. The request is not forwarded to the server-side. Otherwise, the inputs are considered as benign and the request is forwarded to the server-side.

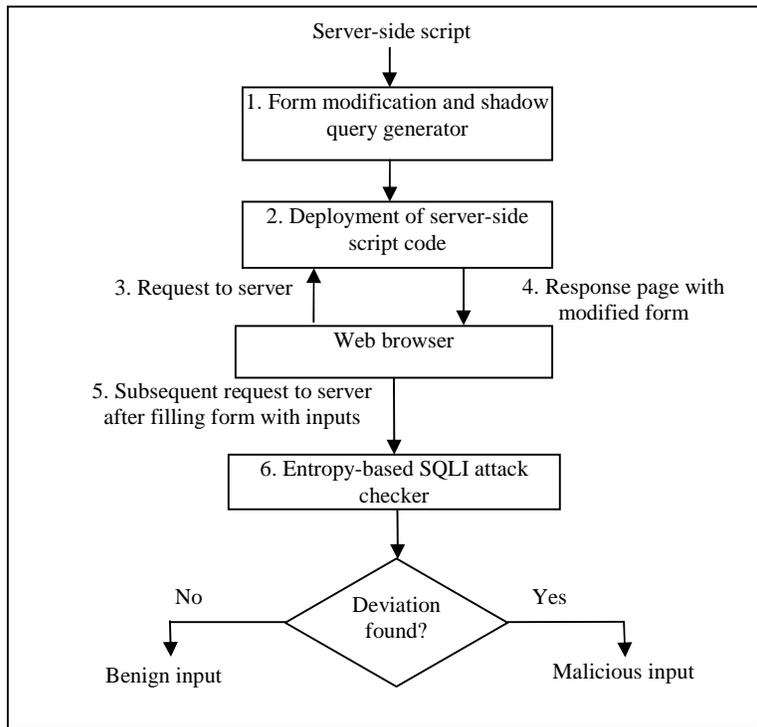


Figure 4. Client-side SQLI attack detection framework.

We now discuss briefly, the shadow query generation and form modification process at the server-side in Subsection 3.1. Then, we discuss the necessity of conditional entropy measures and our proposed metrics to measure query deviation in Subsection 3.2. We also show an example of SQLI attack detection based on conditional entropy metric in Subsection 3.3.

3.1. Shadow query generation and form modification

We first extract the queries by examining the relevant APIs that are related to SQL query execution. In PHP, the *mysql_query()* function call arguments are tracked and if they include other PHP variables, we trace back in the source to see the definition of other variables before concatenating them to form the original query. The query extraction process is approximate (manual intervention may be required for complex code structure). This step becomes challenging

when a query is formed using conditional or loop statement. In this case, our assumption is that the loop runs once. For conditional statement, we identify queries based on the control flow paths. The SQL queries extracted at the server-side script are then used to track the relevant HTML form fields which provide values to relevant PHP variables. We perform a backward static slicing approach [27, 35] to identify whether a form field value becomes part of a query. If it participates, we record and combine all field names followed by placing them in a hidden field of the HTML form.

The extracted queries are simplified to generate shadow queries. A shadow query has the identical structure of an original query, except the column, table, and variable names are replaced with symbolic values. We replace table, column, and variable values with symbolic values based on the following replacement pattern: (i) table names are replaced with t_1, t_2 , and so on; (ii) column names are replaced with c_1, c_2 , and so on; (iii) logical operators are replaced with l_1, l_2 , and so on; (iv) values are replaced with v_1, v_2 , and so on. Note the when substituting query variables that are connected to form fields, we record and store them in a hidden field of the form. The shadow query is sent from the server-side to facilitate the client-side detection.

Once the shadow query is formed and the HTML form is analyzed, we then generate the necessary JavaScript code to compute conditional entropy with the embedded information and detect the deviation between shadow query structure and actual query (formed at the client with supplied input values). In the next section, we provide details of the measurement metrics that are used to identify the deviation.

3.2. Conditional entropy calculation

Conditional entropy is a measurement technique that we brought from information theory [26]. In other words, we are measuring the information content of SQL queries as part of determining the deviation between shadow query and an original query with malicious inputs. Given that we have a set of random variables, and we know the outcome of one variable in advance, the randomness of the remaining variables can be computed with the conditional entropy metric. For a given SQL query, if we have the priori information about the query type (*e.g.*, *select*, *insert*, *delete*), we can measure the information content about certain parts of the query. This allows us to reduce the computation time to determine entropy level.

Table 1 shows the common four types of SQL queries that are dynamically generated at the server-side. We show examples of shadow queries and the selected parts of the queries that we consider for measuring the information content. The last column shows the query part under computation. For example, in the first row, we show a general form of select type shadow query (column, table, field, and value are given symbolic names). The where condition ($f_1=v_1$) is the dynamic part and it is considered for measuring the information content given that we know that the type of query is select. Similarly, for insert type query, we find the values inserted are vulnerable to SQLI attacks. So, we choose the value part of the query to measure the information content. In the same manner, for update type query, we choose set and where condition. Finally, for delete type query, we only consider where condition.

Table 1. Sample query and selected part for conditional entropy computation.

Type	Example query	Selected part	Query part for entropy computation
Select	Select c1, c2 from t1 where c1=v1	Where condition	c1=v1
Insert	Insert into t1 (c1, c2) values (v1, v2)	Value	(v1, v2)
Update	Update t1 set c1=v1 where (c1=v2)	Set, where condition	set c1=v1 where (c1=v2)
Delete	Delete from t1 where c1=v1	Where condition	c1=v1

We now formally present the formula for computing the conditional entropy given that we know the query types.

Let us assume that q be a query present in a program, $T=\{t_1, t_2, \dots, t_M\}$ be the set of tables being used in the query, $C=\{c_1, c_2, \dots, c_L\}$ is the set of all columns that can be selected, inserted, deleted, or updated in q , $V=\{v_1, v_2, \dots, v_P\}$ is the set of values that are used in where conditions or setting values, $L=\{l_1, l_2, \dots, l_P\}$ is the logical operators present in the where condition of q , and $O=\{o_1, o_2, o_3, o_4\}$ is the set of operation types that can be performed by q . Here, o_1, o_2, o_3, o_4 represent *select*, *insert*, *update*, and *delete* operations, respectively.

We now define four different probability and conditional probability functions as follows:

$P(o)$ is the probability that a query q performs specific operation o (*select*, *insert*, *update*, *delete*), where $o \in O$.

$P(c)$ is the probability that column c appears in the query q , where $c \in C$.

$P(v)$ is the probability that a value v appears in a query condition or input value, where $v \in V$.

$P(t)$ is the probability that a query q contains a table t , where $t \in T$.

$P(l)$ is the probability that a query q contains a logical operation (AND, OR, NOT), where $l \in L$.

$P(c/o)$ is the conditional probability of the presence of column c given that we know the operation type o for query q .

$P(v/o)$ is the conditional probability of value v being used in a query's where condition or set values given that we know the operation type of q .

$P(t/o)$ is the conditional probability of table t being accessed or modified in a query given that we know the operation type of q .

$P(l/o)$ is the conditional probability of performing logical operation (in where condition) in q , given that we know the operation type of q .

We now define four conditional entropies of a query q as follows.

Given that we know the operation type of a query, the conditional entropy of column (denoted as H_c) can be computed as follows (Equation (i)):

$$H_c(c/o) = -\sum_{c \in C} P(c, o) \log P(c/o) \dots (i)$$

H_c allows us to detect SQLI attacks where altered query selects additional columns (e.g., *UNION select (1, 1)*).

Given that we know the operation type of a query, the conditional entropy of values (denoted as H_v) can be computed as follows (Equation (ii)):

$$H_v(v/o) = -\sum_{v \in V} P(v, o) \log P(v/o) \dots (ii)$$

H_v allows us to detect attacks where altered query may set new values to fields not intended by the original query (e.g., tautology has $I=I$ in attack signatures where I is assumed as a value).

Given that we know the operation type of a query, the conditional entropy of table (denoted as H_t) can be computed as follows (Equation (iii)):

$$H_t(t/o) = -\sum_{t \in T} P(t, o) \log P(t/o) \dots (iii)$$

H_t allows us to detect attacks where altered query may perform additional operations on table not intended by the original query (e.g., piggybacked queries selecting arbitrary tables).

Given that we know the operation type of a query, the conditional entropy of logical operation (denoted as H_l) can be computed as follows (Equation (iv)):

$$H_l(l/o) = -\sum_{l \in L} P(l, o) \log P(l/o) \dots (iv)$$

H_l allows us to detect attacks where altered query may perform additional logical operations on table not intended by the original query (e.g., tautology).

The unit of the conditional entropy is bit. The minimum value for conditional entropy in a query q can be zero. This might happen if a query accesses just one table, column, or includes one logical condition. There is no upper bound for the conditional entropy value.

3.3. Example of SQLI attack detection with conditional entropy

We show an example application of our proposed conditional entropy metric (H_l) and how it can be used to detect malicious inputs causing SQLI attacks. We reuse the shadow query in Figure 3(a): *select c₁, c₂ from t1 where c₃ = v₁ l₁ c₄ = v₂*. Being a select query, we only consider the where conditions (*c₃=v₁ l₁ c₄=v₂*). We find that

$$L = \{l_1\}, o = \{select\}, P(o) = P(l_1|select) = P(l_1, select) = 1.$$

The conditional entropy for logical operator (denoted as $H_{l-shadow}$) in the shadow query becomes

$$H_{l-shadow}(l/o) = -P(l_1,select) * \log(P(l_1|select)) = -1 * \log(1) = 0.$$

Let us assume that an attacker provides a tautology input (' or l=1 --) in the login field and no input in the password field. Then, where *c₃ = v₁ l₁ c₄ = v₂* becomes *c₃ = ' or l=1 -- l₁ c₄=v₂*. By substituting inputs with symbolic values we obtain *c₃ = ' l₂ c₅=v₁-- l₁ c₄=v₂* (or is replaced with l_2). The revised logical operator space and conditional probabilities are as follows:

$$L = \{l_1, l_2\}, P(l_1|select) = P(l_2|select) = 1/2. P(l_1,select) = P(l_2,select) = 1/2.$$

The conditional entropy for logical operator (denoted as $H_{l-actual}$) becomes

$$H_{l-actual}(l/o) = -P(l_1,select) * \log(P(l_1|select)) - P(l_2,select) * \log(P(l_2|select)) = 1$$

This indicates that the conditional entropy of logical operator increases due to tautology attack. Similarly, we can show the usefulness of the proposed four conditional entropies to detect other types of SQLI attacks. Table 2 shows a mapping between the attack types and the proposed conditional entropy metrics that can be used to detect them at the client side.

Table 2. A mapping between SQLI attack types and conditional entropy metrics.

Attack type	H _c	H _v	H _t	H _l
Tautology				
Union				
Piggybacked query				

4. EVALUATION

We evaluate the proposed conditional entropy-based client-side SQLI attack detection approach using three open source PHP applications available from *sourceforge.net*. The chosen programs have been ranked among the top five most downloaded applications during our evaluation. The programs include *PHP-Address book* (address and contact manger), *Serendipity* (blog management), and *PHP-fusions* (content management system). Table 3 shows some characteristics of these applications that include the number of files we analyze (column 2) and the corresponding lines of code (column 3), the number of forms modified (column 4), and the

number of different query types (column 5-8) related to the form fields. The last two columns show the minimum and maximum number of input fields that we notice in these forms.

We generate the shadow queries and modify forms in PHP applications (all compatible with PHP version 5.3 or above). We deploy the applications in an Apache web server (version 2.2) with MySQL server as the backend database after creating necessary tables with data set. We then visit the web applications so that we can reach the modified pages and access the HTML forms. We supply malicious inputs in form fields. We observe whether the checking at the client-side extension generates a warning and a request is stopped in presence of attack inputs. We apply three types of attack inputs: tautology, union, and piggybacked queries.

Table 4 shows the results obtained during our evaluation with both attack and benign inputs. The second and third columns show the number of attack inputs applied in form field and the corresponding number of warnings generated. The approach successfully detects all the attacks when. Thus, the false negative rate in our evaluation is zero. The fourth and fifth columns show the number of benign input (generated randomly and applied to different form fields) and the number of warning that we notice in our evaluation. The approach does not generate any false positive warning.

Table 3. Characteristics of the applications.

Program	# of files	LOC	# of form	Select	Update	Insert	Delete	Min	Max
PHP-Address book	14	3,422	12	7	2	2	1	2	7
Serendipity	3	5,465	7	4	1	1	1	2	8
PHP-fusions	20	6,419	17	11	3	2	1	2	10

Table 4. Evaluation results.

Program	# of attack inputs applied	# of attacks detected	# of benign inputs applied	# of false warning
PHP-Address book	36	36	240	0
Serendipity	21	21	140	0
PHP fusions	51	51	340	0

Figure 5 shows a snapshot of the time delay overhead due to the computation of four types of conditional entropies for the three applications (we consider the average time taken for entropies for both shadow and actual queries). The x axis shows the four types of conditional entropies and the y axis shows the computation time delays in milliseconds.

The computation of H_c and H_i takes the highest and lowest amount of time for all applications. This is due to the involvement of relatively higher number of columns and lower number of table in queries among these applications. We notice that *PHP-Address Book* application takes more time to compute the conditional entropies than the other two applications. We find that the shadow queries of *PHP-Address Book* is more complex than the other two applications (includes more columns, tables and logical operators).

These contribute to the consumption of the processing time during the attack detection process. Serendipity takes the lowest amount of time during the conditional entropy (except H_i which is the highest among three applications) computation steps. The *Serendipity* application has relatively simple form of queries and hence consume less time during the evaluation. Overall, the

computation time overhead is found to be negligible. We also observe that the amount of HTML code and JavaScript code added due to our approach is also negligible.

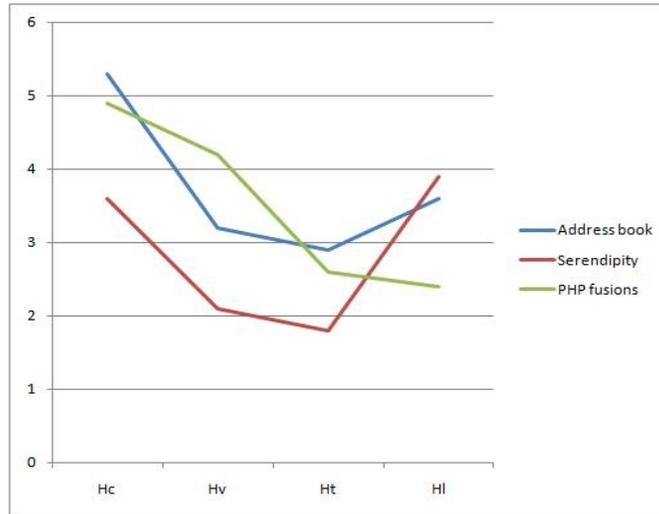


Figure 5. Delay comparison among applications for conditional entropy computation

5. CONCLUSION

SQLI attack is a concern among web application users and a primary source of widespread security breaches. This paper has developed conditional entropy metric-based SQLI attack detection framework based on the priori information supplied from the server-side. Our approach relies on the generation of shadow queries as well as encoding information in HTML forms to enable the necessary checking at the client-side. We measure the deviation between an expected query and an altered query with four conditional entropy metrics that are capable of detecting different types of known and unknown attacks. The approach not only allows detection of malicious inputs causing SQLI at the client-side early, but also relieves the server-side for additional checking and acts as a complementary solution to other existing approaches. We evaluate the proposed approach with three PHP applications containing known SQLI vulnerabilities. The results indicate that the approach can successfully stop malicious inputs at the client-side without any significant overhead.

Our future plan includes developing more metrics to consider the deviation of complex form of SQL queries that can be altered under attack inputs and SQLI attacks on stored procedures. Future work also includes evaluating our approach with more open source web applications. Moreover, we are planning to apply the concepts from information theory to mitigate other web-based attacks such as cross-site scripting.

ACKNOWLEDGEMENTS

This work was funded in part by Faculty Summer Research Award, College of Science and Mathematics, Kennesaw State University, USA. The authors would like to thank reviewers for their thoughtful comments to improve this article.

REFERENCES

- [1] The Open Web Application Security Project (OWASP), https://www.owasp.org/index.php/Top_10_2010-Main.
- [2] G. Buehrer, B. W. Weide, P. A. G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05), Lisbon, Portugal, 2005, pp.106-113.
- [3] . Boyd and A. Keromytis, "SQLrand: Preventing SQL injection attacks", Proc. of the 2nd Applied Cryptography and Network Security Conference, June 2004, pp. 292-302.
- [4] Z. Su and G. Wasserman, "The Essence of Command Injection Attacks in Web Applications," Proc. of the Symposium on Principles of Programming Languages (POPL), January 2006, South Carolina, USA, pp. 372-382.
- [5] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, Y. Takahama, "Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection," Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC), Miami, Dec 2007, pp. 107-117.
- [6] Open Source Vulnerability Database, <http://osvdb.org>
- [7] Y. Shin, L. Williams, and T. Xie, "SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis," Proc. of the International Symposium on Software Reliability Engineering (ISSRE), Raleigh, NC, 2006, ISBN 978-0-9671473-3-3-8.
- [8] S. Curtis, "Barclays: 97 percent of data breaches still due to SQL injection", January 2012, <http://news.techworld.com/security/3331283/barclays-97-percent-of-data-breaches-still-due-sql-injection>
- [9] S. Thomas and L. Williams, "Using Automated Fix Generation to Secure SQL Statements," Proc. of the 3rd International Workshop on Software Engineering for Secure Systems (SESS'07), Minneapolis, 2007, pp. 9-14.
- [10] S. Bandhakavi, P. Bisth, P. Madhusudan, and V. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," Proc. of the 14th ACM Conf. on Computer and Communications Security, Alexandria, Virginia, Oct 2007, pp. 12-24.
- [11] J. Lin and J. Chen, "The Automatic Defense Mechanism for Malicious Injection Attack," Proc. of the 7th International Conference on Computer and Information Technology, Fukushima, Japan, October 2007, pp. 709-714.
- [12] Web Application Security Consortium, <http://www.webappsec.org>
- [13] K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL-Injection Detection", Proc. of 23rd ACM Symposium on Applied Computing (SAC'08), March 2008, Fortaleza, Brazil, pp. 2153-2158.
- [14] SQL-inject-me, <https://addons.mozilla.org/en-us/firefox/addon/sql-inject-me/>
- [15] W3af, Open Source Web Application Security Scanner, <http://w3af.org>
- [16] W. Lee and D. Xiang, "Information-Theoretic Measures for Anomaly Detection," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, May 2001, pp. 130-143.
- [17] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," Proceedings of the 8th IEEE International Conference on Quality Software (QSIC), London, UK, August 2008, pp. 77-86.
- [18] E. Allen and T. Khoshgoftaar, "Measuring Coupling and Cohesion: An Information Theory Approach," Proceedings of the 6th International Software Metrics Symposium, November 1999, Boca Raton, FL, pp. 119-127.
- [19] Sqlmap, <http://sqlmap.org/>
- [20] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skoric, "Towards an Information-Theoretic Framework for Analyzing Intrusion Detection Systems," Proc. of the 11th European Symposium Research Computer Security, Hamburg, Germany, Sept2006, pp. 527-546.
- [21] H. Shahriar and M. Zulkernine, "Mitigation of Program Security Vulnerabilities: Approaches and Challenges," ACM Computing Surveys (CSUR), Vol. 44, No. 3, Article 11, May 2012.
- [22] Y. Geng, "An information-theoretic model for resource-constrained systems," Proc. of IEEE International Conference on Systems Man and Cybernetics (SMC), October 2010, Istanbul, Turkey, pp. 4282-4287.

- [23] S. Khayam, H. Radha, and D. Loguinov, "Worm Detection at Network Endpoints Using Information-Theoretic Traffic Perturbations," Proc. of the IEEE International Conference on Communications (ICC), Beijing, China, May 2008, pp. 1561-1565.
- [24] H. Shahriar and M. Zulkernine, "Information Theoretic Detection of SQLI Attacks," Proc. of 14th IEEE International High Assurance Systems Engineering Symposium (HASE), Omaha, Nebraska, USA, October 2012, pp. 40-47.
- [25] T. Cover and J. Thomas, Elements of Information Theory, John Wiley and Sons, 2006.
- [26] C. Shannon, "A Mathematical Theory of Communication," Bell Systems Technical Journal, 1948.
- [27] F. Tip, A Survey of Program Slicing Techniques, Technical Report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1994.
- [28] A. Singh and S. Roy, "A Network Based Vulnerability Scanner for Detecting SQLI Attacks in Web Applications," Proc. of the 1st International Conference on Recent Advances in Information Technology (RAIT), March 2012, Dhanbad, India, pp. 585-590.
- [29] G. Agosta, A. Barenghi, A. Parata, G. Pelosi, "Automated Security Analysis of Dynamic Web Applications through Symbolic Code Execution," Proc. of the 9th International Conference on Information Technology: New Generations (ITNG), April 2012, Las Vegas, NV, pp. 189-194.
- [30] J. Kim, "Injection Attack Detection Using the Removal of SQL Query Attribute Values," Proc. of the International Conference on Information Science and Applications (ICISA), Jeju Island, Korea, May 2011, pp. 1-7.
- [31] N. Antunes and M. Vieira, "Defending Against Web Application Vulnerabilities," IEEE Computer, Volume 45, Issue 2, February 2012, pp. 66-72.
- [32] SqliFuzzer, Command Line SQL Injection Web Scanner, <http://code.google.com/p/sqlifuzzer>
- [33] M. Johns, C. Beyerlein, R. Giesecke, J. Poesgga, "Secure Code Generation for Web Applications," Proc. of the 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10), Pisa, Italy, LNCS 5965, pp. 96-113, Springer.
- [34] H. Shahriar, S. North, and W. Wei, "Client-Side Detection of SQL Injection Attacks," Proc. of 3rd International Workshop on Information Systems Security Engineering (WISSE), X. Franch and P. Soffer (Eds.): CAiSE 2013 Workshops, LNBIP 148, pp. 512-517, Barcelona, Spain, June 2013. Springer, Heidelberg (2013).
- [35] M. Weiser, "Program Slicing," IEEE Transactions ON Software Engineering, Vol. SE-10, No. 4, July 1984, pp. 352-357.

Authors

Dr. Hossain Shahriar is currently an Assistant Professor of Computer Science at Kennesaw State University, Georgia, USA. His research interests include web application and network security. Dr. Shahriar has published many research articles in Journals and conferences including *Journal of Systems and Software* and *Future Generation Computer Systems*. He served as reviewer of international journals and conferences related to software and web application security. Currently, Dr. Shahriar is a member of the ACM and IEEE. More information about his research and background can be found at <http://cs.kennesaw.edu/hshahria/>



Dr. Sarah M. North is an Assistant Professor of Computer Science Department at Kennesaw State University. Dr. North has been successfully involved in the research in the areas of information security education, human-computer interaction and cognitive science. Dr. North has published several book chapters; and a number of referred scholarly articles in national and international venues. She also served as principal/co-principal investigator on a number of research grants sponsored by the Boeing Company, National Science Foundation, and National Security Agency. Dr. North 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>



Wei-Chuen Chen is currently an undergraduate student at Kennesaw State University. His interests include security in computing and data analysis. During the summer of 2013, he held an intern position at Oversight Systems where the company product provides continuous transaction analysis for big data. Currently, he is a member of ACM and an officer of KSU ACM student chapter.

