Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

12-2012

Semi-automated verification of defense against SQL injection in web applications

Kaiping LIU
Nanyang Technological University

Hee Beng Kuan TAN
Nanyang Technological University

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Information Security Commons, and the Software Engineering Commons

Citation

LIU, Kaiping; TAN, Hee Beng Kuan; and SHAR, Lwin Khin. Semi-automated verification of defense against SQL injection in web applications. (2012). *2012 19th Asia-Pacific Software Engineering Conference (APSEC): Hong Kong, December 4-7: Proceedings*. 91-96.
Available at: https://ink.library.smu.edu.sg/sis_research/4838

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Published in 2012 19th Asia-Pacific Software Engineering Conference (APSEC): Hong Kong, China, December 4-7: Proceedings. pp. 91-96.
https://doi.org/10.1109/APSEC.2012.18

Semi-Automated Verification of Defense against SQL Injection in Web Applications

Kaiping Liu
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore
kpliu@ntu.edu.sg

Hee Beng Kuan Tan
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore
ibktan@ntu.edu.sg

Lwin Khin Shar
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore
Shar0035@ntu.edu.sg

***Abstract*—Recent reports reveal that majority of the attacks to Web applications are input manipulation attacks. Among these attacks, SQL injection attack – malicious input is submitted to manipulate the database in a way that was unintended by the applications' developers – is one such attack. This paper proposes an approach for assisting to code verification process on the defense against SQL injection. The approach extracts all such defenses implemented in code. With the use of the proposed approach, developers, testers or auditors can then check the defenses extracted from code to verify their adequacy. We have evaluated the feasibility, effectiveness, and usefulness of the proposed approach by a set of open-source systems. Our experiment results showed that the proposed approach is effective in extracting all the possible defenses implemented/adopted by Web applications. We observed that the proposed approach would be useful in identifying the false positive cases resulting from other related approaches and auditing the code in order to fix the actual vulnerable cases.**

Keywords-SQL injection, vulnerabilities, code auditing, software security, static analysis, Web applications

I. INTRODUCTION

Recent reports on security attacks consistently showed that most security attacks to Web applications are not caused by break-through encryption mechanisms or by hacking network security protocols. They are caused by illegal input manipulation – hackers enter inputs that are not intended to be processed by a system to achieve their purpose. Such attacks may lead to unauthorized access to sensitive data, insertion, modification or deletion to database. This is the SQL injection vulnerability (SQLIV), which has been ranked among the top ten vulnerabilities over the past few years due to its popularity and severity [1]. Reports on SQL injection attacks (SQLIA) showed that they are mainly performed through illegal input manipulation due to code vulnerability [2]. As an illustration, a code snippet vulnerable to SQL injection is shown in the following:

```
httpSeverletRequest request = .....;
String accountCode =
    request.getParameter("acoountCode");
Connection con = .....;
String query = "SELECT * FROM Accounts WHERE
    accountNo = '" + accountCode + " '";
con.execute(query);
```

A hacker may enter "`' OR 1 = 1`" as an input to accountCode in order to produce the query string as "`SELECT * FROM Accounts WHERE accountNo = '' OR 1 = 1`". As a result, the where-clause of the query becomes a tautology. This allows the hacker to bypass the account code check and get access to all account records in the database.

Traditionally, input validation and input sanitization are used to defend SQLI. In this paper, we propose a novel code verification method which takes a different approach from existing methods. Based on the possible coding patterns for implementing defense against SQL injection, the proposed approach automatically extracts all the possible such defenses from source code by performing static analysis. The extracted output can then be checked to verify the adequacy and identify the potential risks. It is also possible that existing static analysis approaches could also be incorporated into our work in order to automate both extraction of SQL injection defenses and detection of SQLIVs.

II. THEORY FOR EXTRACTING DEFENSE AGAINST SQL INJECTION FROM CODE

In a Web application, a node u in a control flow graph (CFG) such that an input submitted by user can be referenced at u and u dominates all nodes w at which the input can also be referenced, is called an input node. A variable in the input is called an input variable submitted at u. A path through a CFG is called a 1-path if it follows any loop at most one time (that is, if it does not repeat any loop). Let w be an input node in a CFG. An input path of w is a path from w to the exit node that does not pass through w again. We may also call an input path of an input node simply, an input path depending on the context used. An input path of w is called a prime input path of w if it iterates any loop at most one time.

In a Web application, a statement in a program that performs a SQL operation is called an SQL operation statement (sql-o-statement). The node in the CFG of the program that represents the statement is called an SQL operation node (sql-o-node). We shall also adopt some formalism of control flow graph from [3], including dominance and transitive dominance.

```
1.    String formAction = request.getParameter(“formAction”);
2.    String userid = request.getParameter(“userid”);
3.    String password = request.getParameter(“password”);

4.    String sQuery= “SELECT * FROM customer ”;
5.    String sWhere= “” ;
                . . .
6.    Connection con = . . . ;

7.    if (formAction.equals(“Login”) {
8.          password.replace(“ ‘ ”, “ ‘’ ”);
9.          if ((!userid.equals(“”)) && (!password.equals(“”))) {
10.               if (isNum(userid)) {
11.                   sWhere= “WHERE userid =“+userid+” AND
                        password = ' “+password+ “ ' ”;
12.                   con.executeQuery(sQuery+sWhere);
13.                   session.setAttribute(“UserID”, userid);
                                . . .
                  }
          }
          else {
14.               response.sendRedirect(“login.html”);
          }
      }
15.   if (formAction.equals(“Logout”) {
16.         session.setAttribute(“UserID”, “”);
            . . .
      }
```

Figure 1. JSP code snippet for authentication

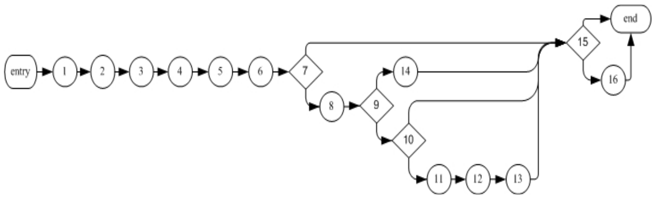

Figure 2. The CFG of the JSP code snippet for authentication

A. *Extracting Defense through Input Validation*

The proposed approach will extract the statements that could be for the purpose of defending against SQL injection. Let k be a sql-o-node in a CFG. To prevent input operated at k from illegal manipulation that may lead to SQL injection attack, one must ensure that only input that satisfies the required condition will be operated at k through the use of predicate node. Next, we shall define a terminology to characterize such node pattern.

A predicate node d is called a validation node for k if the following properties hold:

1) Both k and d transitively reference to a common input variable submitted at an input node w.
2) There is a prime input path p of w that follows one branch of d passes through k and there is no prime input path p' of w that follows the other branch of d passes through k.

In Fig. 2, both the sql-o-node, node 12, and the predicate node, node 9, transitively reference to the input variable, `password`, submitted at the input node, node 3. The prime input path (3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, end) that follows the branch (9, 10) of node 9 passes through node 12. No prime input path that follows the other branch (9, 14) of node 9 passes through node 12. Hence, node 9 is a validation node for the sql-o-node, node 12.

Property 1 – Unprotected Sql-O-Node. Let k be a sql-o-node in a CFG. If k transitively references to an input variable v submitted at an input node w and there is no validation node for k that transitively references to v, then k is unprotected from SQL injection that may arise from manipulating value of v.

Let k be a sql-o-node in a CFG. Let Ω be the set of 1-paths through the CFG. The partition of the 1-paths in Ω, such that paths, which pass through the same set of validation nodes for k and follow the same branch at each of these nodes, are put in the same class, is called the **validation partition** for k.

In the CFG shown in Fig 2, {{(entry, 1, 2, 3, 4, 5, 6, 7, 15, end), (entry, 1, 2, 3, 4, 5, 6, 7, 15, 16, end)}, {(entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 15, end), (entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 15, 16, end)}, {(entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, end), (entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16, end)}, {(entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, end), (entry, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 16, end)}} is the validation partition for the sql-o-node, node 12, in this CFG.

An input condition that will lead to the execution of a sql-o-node is called a **valid input condition of the sql-o-node**. An input condition that will not lead to the execution of a sql-o-node is called an **invalid input condition of the sql-o-node**.

Property 2 – Invalid Input Condition. Let k be a sql-o-node k in a CFG. The set of invalid input conditions for k is IVC_k = {C ∣ X is a class in the validation partition of k such that there is a path in X passes through some input validation nodes for k and does not pass through k; and C = the conjunction of all the branch conditions of branches at validation nodes for k that any path in X follows}.

Property 3 – Valid Input Condition. Let k be a sql-o-node k in a CFG. The set of valid input conditions for k is VC_k = {C ∣ X is a class in the validation partition of k such that there is a path in X passes through some input validation nodes for k and also passes through k; and C = the conjunction of all the branch conditions of branches at validation nodes for k that any path in X follows}.

For the program with the CFG shown in Fig. 2, we have already computed the validation partition for the sql-o-node, node 12. From Property 2, IVC = {!((!userid.equals(“”)) && (!password.equals(“”))), ((!userid.equals(“”)) && (!password.equals(“”))) && (!(isNum(userid)))} is the set of invalid input conditions for the sql-o-node. From Property 3, {((!userid.equals(“”)) && (!password.equals(“”))) && (isNum(userid))} is the set of valid input conditions for the sql-o-node.

B. *Extraction of Defense through Input Filtering*

Let v be a variable referenced at a sql-o-statement. Only if v falls under one of the following two cases, then it may be possible for v to be manipulated by a hacker:

1) v is submitted at an input node.
2) v is defined at a node that transitively reference to input variables submitted at input nodes.

Hence, a hacker can only manipulate a variable referenced in a sql-o-statement that falls under one of the above-mentioned condition to attack a Web application by SQL injection through input manipulation. We call such variable, a **potentially vulnerable variable** (**pv-variable**) of the sql-o-statement. For example, in Fig. 1, `userid` and `password` are pv-variables of the sql-o-statement shown at line 12.

We call a sequence of all the input filtering statements according to their order in the program, the potentially vulnerable input filter (or pv-input filter) for the sql-o-statement. More formally, the pv-input filter for a sql-o-statement k in program P is a sequence F of following statements according to their order in P:

1) All the statements at which a pv-variable of k is defined/submitted.
2) Statements on which statements in F are data dependent.
3) Statements on which statements in F are control dependent such that k is not transitively control dependent on these statements.

In the program shown in Fig. 1, the sequence of the following statements shown at line 2, 3, 8, 11 forms the pv-input filter for the sql-o-statement shown at line 12:

```
2.  String userid =
     request.getParameter("userid");
3.  String password =
     request.getParameter("password");
8.  password.replace(" ' ", " \' ");
11. sWhere= "WHERE userid ="+userid+" AND
     password = ' "+password+ " ' ";
```

III. The Proposed Code Verification Approach

Building on the theory discussed in previous section, the proposed approach checks the defense against SQL injection implemented in Web applications through the following two major steps:

Step 1: Extract the defense against SQL injection implemented automatically from code.

Step 2: Examine the extracted output to verify its adequacy.

The algorithm for implementing the first step is shown in Fig. 3. For each sql-o-statement *k* in *P*, the information on defense through input validation for *k* is extracted based on the theory discussed in Section 2.1 from the CFG, *G*, of the program according to the algorithm shown in Fig. 4.

The algorithm in Fig. 5 shows the computation of pv-input filter F_k for a sql-o-statement *k*. First, we include statements in *P* at which any pv-variable of *k* is defined/submitted in F_k in the same order as they appear in *P*. Then, each time, *P* is processed iteratively to include more statements in F_k until F_k is stabilized – that is no further statements can be included in F_k in an iteration. In each iteration, statements in *P* are processed from the first statement to the last statement sequentially as follows. For each statement *s* in *P*, *s* in included in F_k if it satisfies one of the following conditions:

1) If there is a statement in F_k that is data-dependent on *s*.
2) If *s* is a predicate node, *k* is not transitively control dependent on *s* and there is a statement in F_k that is control dependent on *s*.

From the defense against SQL injection extracted in Step 1, the second step of the proposed approach examines whether the input validation and input filtering implemented for each sql-o-statement is sufficient for defending it against SQL injection. A sql-o-statement could be defended through input validation, input filtering or a combination of them.

In the examination of the defense through input validation, from the output extracted in Step 1, those sql-o-statements that are unprotected from SQL injection through some input variables are clearly without any defense through input validation with regards to these input variables. For each of the remaining sql-o-statements, we examine the invalid and valid input conditions of the sql-o-statement according to the input format and system requirements to examine its adequacy through input validation.

In the examination of the defense through input filtering, from the pv-input filter for a sql-o-statement extracted from Step 1, one needs to examine how the pv-input filter contributes to the defense against SQL injection for the sql-o-statement. Similarly, program slicing can be used to aid for the comprehension of the pv-input filter through slicing on its variables and associated statements.

```
Algorithm extractDefenseSqlInj(P: program)
Output: The set $\Pi$ of defense against SQL injection for each sql-o-
        statement in P.
begin
1.  initialize both F and D to empty sets;
2.  compute G = the CFG of P:
3.  for (each sql-o-node k in G) do
4.      $D_k$ = extractInputValidation(G; k);
5.      include the tuple (k, $D_k$) in D;
6.      $F_k$ = extractPvInputFilter(P, k);
7.      include the tuple (k, $F_k$) in F;
    endFor;
8.  $\Pi$ = (D, F);
9.  return $\Pi$;
end;
```

Figure 3. Algorithm for extraction of defense against SQL injection

IV. EVALUATION

To evaluate the proposed approach, we have developed a prototype tool called SQLIDE (SQL Injection Defense Extractor) to implement the algorithm shown in Fig. 3 discussed in Section 3. With the use of prototype tool, we have evaluated the feasibility, effectiveness, and usefulness of the proposed approach on seven open source systems. In our evaluation, we have compared our approach with the approach proposed by Livshits and Lam [4] that is most commonly referenced work for the detection of security flaws in code.

A. Prototype Tool

The prototype tool SQLIDE is developed through the use of the Java Architecture for Bytecode Analysis (JABA) from Georgia Institute of Technology [5] for Web-based database applications written in Java. It consists of two major modules: a program analyzer, and an input validation and filtering miner (IVF miner). Program analyzer uses JABA's APIs to analyze Java programs. It takes the class files of a Java program as input and builds the CFG of the program for control flow and data flow analysis. IVF miner includes three sub-modules: an input validation extractor (IV extractor), a pv-input filter extractor (PVIF extractor), and a program slicer.

B. Experiment Results

Table I gives an overview of the applications experimented. We evaluate the most precise analysis by enabling both context sensitivity and improved object naming.

The statistics of the results are shown in Table II and III respectively. From the tables, we can see that our proposed approach achieve zero false positive while Livshits and Lam's approach produces 34.47% false positive rate. Our proposed approach is also more effectiveness in detecting defense of SQL injections. Though the proposed approach extracts more program artifacts, we observed that manual verification process in identifying the SQLIVs is still feasible for the sizes of the experimented applications because the two students completed the experiment without any problem. More importantly, the students also reported that comprehending the vulnerability is made easier with all the implemented SQL injection defense features extracted.

```
Algorithm extractInputValidation(G: the CFG of a program; k: sql-o-node in G)
Output: A tuple $D_k$ with three elements, $UP_k$, $IVC_k$ and $VC_k$, where $UP_k$ =
        {(v, w) | k is unprotected from SQL injection through input
        variable v submitted at input node w}, $IVC_k$ is the set of invalid
        input conditions for k and $VC_k$ is the set of valid input
        conditions for k.
begin
1.  compute V = the set of validation node for k;
2.  compute $\Omega$ = the set of 1-path through G;
3.  compute $\Phi$ = the input validation partition for k;
4.  for (each input variable v submitted at input node w that is transitively
    referenced at k) do
5.      If (there is no input validation node for k that that transitively
            references to v) then
6.          include (v, w) in $UP_k$;
        endIf;
    endFor;

7.  for (each X in $\Phi$) do
        p = a path in X;
8.      if (p passes through an input validation node for k) then
9.          if (p does not pass through k) then
10.             include the conjunction of all the branch conditions
                    of branches at validation nodes for k that p
                    follows in $IVC_k$;
            else
11.             include the conjunction of all the branch conditions
                    of branches at validation nodes for k that p
                    follows in $VC_k$;
            endIf;
        endIf;
    endFor;
12. $D_k = (UP_k, VC_k, IVC_k)$;
13. return $D_k$;
end;
```

Figure 4. Algorithm for extraction of defense through input validation for a sql-o-node

Table I. Overview of Web Applications experimented

Application	Description	LOC	No. of Servlets	No. of sql-o-statements
Employee Directory	Online employee directory	3,035	10	19
Bookstore	Online bookstore	9,551	28	71
Events	Event tracking system	3,818	13	25
Classifieds	Online management system for classified	5,745	19	43
Portal	Portal for a club	8,803	28	60
Roomba	Online hotel room booking system	10,251	39	158
Smacs	Online management of casual staff	5,574	24	41
Total		**46,777**	**161**	**417**

```
Algorithm extractPvInputFilter(P: program, k: sql-o-statement in P)
Output: The pv-input filter $F_k$ for k.
begin
1.  initialize both F and F' to empty sequences of statements;
2.  include statements in P at which any pv-variable of k is
        defined/submitted in F according to their order in P;
3.  while ($F_k \neq F'$) do
4.      $F' = F_k$;
5.      for (each statement s in P from the first statement until
            the last statement) do
6.          if s is not a predicate node then
7.              if (there is a statement in $F_k$ data-dependent
                on s) then
8.                  include s in $F_k$;
                endIf;
            else
9.              if (k is not transitively control dependent on s)
                then
10.                 if (there is a statement in $F_k$ that is
                    control-dependent on s) then
11.                     include s in $F_k$;
                    endIf;
                endIf;
            endIf;
        endFor;
    endWhile;
12. return $F_k$;
end;
```

Figure 5. Algorithm for extraction of pv-input filter for a sql-o-statement

Table II. Statistics of evaluation results for Livshits and Lam's approach

Application	#. vulnerable sql-o-statement	Traces extracted (LOC)				False positives	# of actual vulnerable sql-o-statment
		From tool		Confirmed for defense against SQL injection			
		Total	Average per servlet	Total	Average per servlet		
Emp. Dir.	12	83	8.3	56	5.6	4	8
Book-store	38	279	9.96	181	6.46	21	17
Events	16	107	8.23	72	5.53	6	10
Classi-fieds	34	186	9.78	110	5.78	17	17
Portal	34	264	9.42	178	6.35	10	24
Roomba	61	261	6.69	137	3.51	6	55
Smacs	40	236	9.83	83	3.45	17	23
Total	**235**	**1,416**	**8.79**	**817**	**5.07**	**81**	**154**

In summary, it is observed that the proposed approach would be especially useful in identifying the false positive cases resulting from other related static analysis approaches and verifying the code in order to fix the actual vulnerable cases.

V. Related Work

A. *Detection of SQL Injection Vulnerabilities*

Approaches in this area are mainly based on static program analysis techniques [4, 6, 7, 8, 9, 10, 11]. Some of them may be augmented with dynamic analysis [6], string analysis [6, 9, 10], symbolic execution [11] or alias analysis [8]. Most of these approaches make inference on existence of security vulnerabilities based on the following information: (1) user specification on vulnerability patterns in terms of the flow of external inputs to SQL statements [4]; (2) inadequate or absence of sanitization mechanisms [6, 10, 11, 12]; and (3) potential type mismatch [7] or syntax mismatch [9] between a set of possible SQL strings due to external inputs and the original SQL statement intended by developer. However, most of these approaches are control flow insensitive [4, 8, 9]. Hence, false positive cases would occur if the 'if'-constructs implemented by the programs could effectively prevent the external inputs from injecting the malicious characters into SQL statements. Most of them also do not check custom sanitization functions [4, 8, 9, 11] but only make conservative assumptions. As a result, relatively high false positive rate would be introduced.

Table III. Statistics of evaluation results for the proposed approach

Application	No. of predicate nodes from which valid and invalid input conditions are extracted				Pv-input filter extracted (LOC)				No. of vulnerable sql-o-statements
	From tool		Confirmed for defense against SQL injection		From tool		Confirmed for defense against SQL injection		
	Total	Average per servlet	Total	Average per servlet	Total	Average per servlet	Total	Average per servlet	
Emp. Dir.	25	2.5	25	2.5	164	16.4	131	13.1	8
Bookstore	112	4	112	4	349	12.46	224	8	17
Events	39	3	39	3	135	10.38	87	6.69	10
Classifieds	57	3	57	3	251	13.2	154	8.10	17
Portal	72	2.57	72	2.57	394	14.07	222	7.92	24
Roomba	30	0.76	27	0.69	316	8.10	145	3.71	55
Smacs	39	1.62	39	1.62	272	11.33	139	5.79	23
Total	**374**	**2.32**	**371**	**2.30**	**1,881**	**11.68**	**1,102**	**6.84**	**154**

However, in contrast to the proposed approach, most of the above approaches only highlight vulnerable SQL statements without providing much further information. Although some approaches provide more information to the testers, they mainly show only the data flow traces (i.e., statements on which variables referenced in vulnerable SQL statements are directly or indirectly data dependent on) [4, 8, 9, 10].

B. Input Validation and Security Testing Approach

Input validation is the key to enforce input accuracy. It is also a key to defend security attacks against Web applications. Both specification-based and code-based approaches have been proposed for testing the adequacy of input validation schemes [13, 14, 15, 16, 17, 18, 19]. Specification-based input validation testing approaches generate test cases with the aim of exercising valid and invalid input conditions as complete as possible [13, 14, 15, 18]. To avoid the sole dependency on user specifications, Li et al. [15] augmented the traditional specification-based strategy with automated extraction of input specification through analyzing the HTML pages. However, all these approaches' ability to detect SQL injection vulnerabilities still mainly depends on the completeness of user specifications and the adequacy of test suite generated.

Works on the area of SQLIV testing approach mainly involve injecting attack vectors into the application under test in order to expose SQLIVs [20, 21, 22, 23, 24, 25]. Antunes et al. [21] learn the legitimate syntax structures of SQL queries through the executions of valid test inputs and compare them with the syntax structures of SQL queries resulting from executions of test inputs containing SQL injection attack vectors. If there is any mismatch, the vulnerability of that particular SQL query is detected. Some of the other approaches first inject the vulnerable SQL queries into the application under test and next exercise test inputs that contain SQL injection attack vectors [20, 22].

The common major disadvantage of the above attack vector injection approaches is that there may be false negative cases if the source or the library, which is used to generate SQL injection attack vectors, is incomplete or imperfect. In that case, the sanitization routines implemented by the programs might prevent all the attacks generated by these approaches. But in actual real life attacks, some sophisticated attack vectors may succeed in circumventing those sanitization routines. Similarly to some approaches discussed in Section 5.1, the above approaches only show very limited information regarding to the SQL injection defenses implemented in the programs. Although the purpose of the approach in [22] shares with that of our proposed approach, it only provides information of attack vectors and exploited vulnerabilities, which is also the case of other approaches [20, 21, 23, 24, 25]. Since no information on the security defense features is given, they are not suitable for assisting SQLIV verification process.

C. Prevention of Security Attacks

Works on this area mainly incorporate dynamic monitoring systems into server programs to ensure that the syntax of a dynamic SQL statement built using user inputs as parameters follows the intended structure defined in program before any execution [26, 27, 28, 29, 30, 31]. Their approaches learn the valid query syntax of each SQL statement based on the following ways: (1) static analysis [26]; (2) dynamic taint analysis [27, 28, 29]; or (3) user specification [30, 31]. Their main drawbacks are: (1) they introduce additional overhead into a program for the runtime check; (2) the instrumentation of checking mechanism may introduce further complexity to the debugging of security vulnerabilities. However, all these approaches only serve to the protection of the deployed systems from SQL injection attacks. In contrast, our proposed approach intends to assist the developers or testers in verifying or fixing the vulnerable pieces of codes during the implementation stage.

VI. CONCLUSION

SQL injection is one of the common security threats to Web applications. Fixing or debugging the actual cases would require verification on the adequacy of already implemented SQL injection defenses. However, code verification on the whole source code or on the limited information provided by existing approaches would be either labor-intensive or inadequate. Thus, we have proposed a semi-automated approach for extracting all the defenses against SQL injection implemented in code to facilitate the verification process and also identify the SQL injection vulnerabilities based on inadequate SQL injection defenses. The approach has been evaluated based on the applications that are commonly used for evaluating related approaches. Results have shown that the proposed approach is feasible for the sizes of experimented applications and useful in assisting to SQL injection vulnerability verification process due to its effectiveness in extracting all the statements relevant to SQL injection defenses. For our future work, we intend to evaluate the feasibility and usefulness of the proposed approach based on intensive experiments on both open-source and industrial Web applications of larger sizes.

References

[1] OWASP. The Ten Most Critical Web Application Security Vulnerabilities. [Online]. Available: http://www.owasp.org/index.php/-Category:OWASP_Top_Ten_Project

[2] C. Anley, "Advanced SQL Injection in SQL Server Applications," *White paper, Next Generation Security Software Ltd*, 2002.

[3] S. Sinha, M. Harrold, and G. Rothermel, "Interprocedural Control Dependence," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 209–254, 2001.

[4] V. Livshits and M. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium-Volume 14*. USENIX Association, 2005, pp. 18–18.

[5] A. R. Group *et al.*, "JABA: Java Architecture for Bytecode Analysis," 2003. [Online]. Available: http://www.cc.gatech.edu/aristotle/Tools/-jaba.html

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," in *Proceedings of IEEE Symposium on Security and Privacy, 2008*. IEEE, 2008, pp. 387–401.

[7] C. Gould, Z. Su, and P. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE, 2004, pp. 645–654.

[8] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in *Proceedings of IEEE Symposium on Security and Privacy, 2006*. IEEE, 2006, pp. 6–pp.

[9] G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 32–41.

[10] G. Wassermann and Z. Su, "Static Detection of Cross-site Scripting Vulnerabilities," in *In Proceedings of ACM/IEEE 30th International Conference on Software Engineering, 2008*. IEEE, 2008, pp. 171–180.

[11] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," in *15th USENIX Security Symposium*, 2006, pp. 179–192.

[12] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 1293–1296.

[13] J. Hayes and A. Offutt, "Increased Software Reliability Through Input Validation Analysis and Testing," in *Proceedings of 10th International Symposium on Software Reliability Engineering, 1999*. IEEE, 1999, pp. 199–209.

[14] J. Hayes and J. Offutt, "Input Validation Analysis and Testing," *Empirical Software Engineering*, vol. 11, no. 4, pp. 493–522, 2006.

[15] N. Li, J. Wu, M. Jin, and C. Liu, "Web Application Model Recovery for User Input Validation Testing," in *Proceedings of International Conference on Software Engineering Advances, 2007*. IEEE, 2007, pp. 13–13.

[16] H. Liu and H B K. Tan, "Testing Input Validation in Web Applications Through Automated Model Recovery," *Journal of Systems and Software*, vol. 81, no. 2, pp. 222–233, 2008.

[17] H. Liu and H B K. Tan, "Covering Code Behavior on Input Validation in Functional Testing," *Information and Software Technology*, vol. 51, no. 2, pp. 546–553, 2009.

[18] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass Testing of Web Applications," in *15th International Symposium on Software Reliability Engineering, 2004*. IEEE, 2004, pp. 187–197.

[19] H. Liu and H B K. Tan, "An approach for the maintenance of input validation," *Information and Software Technology*, vol. 50, no. 5, pp. 449–461, 2008.

[20] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," in *In Proceedings of The 8th International Conference on Quality Software, 2008*. IEEE, 2008, pp. 77–86.

[21] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services," in *Proceedings of IEEE International Conference on Services Computing, 2009*. IEEE, 2009, pp. 260–267.

[22] J. Fonseca, M. Vieira, and H. Madeira, "Vulnerability & Attack Injection for Web Applications," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks, 2009*. IEEE, 2009, pp. 93–102.

[23] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst, "Automatic Creation of SQL Injection and Cross-site Scripting Attacks," in *Proceedings of IEEE 31st International Conference on Software Engineering, 2009*. Ieee, 2009, pp. 199–209.

[24] M. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing Web Applications with Static and Dynamic Information Flow Tracking," in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2008, pp. 3–12.

[25] M. Martin and M. Lam, "Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking," in *Proceedings of the 17th Conference on Security Symposium*. USENIX Association, 2008, pp. 31–43.

[26] W. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2005, pp. 174–183.

[27] W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-aware Evaluation," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 65–81, 2008.

[28] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 372–382.

[29] G. Buehrer, B. Weide, and P. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*. ACM, 2005, pp. 106–113.

[30] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in *Proceedings of the 13th International Conference on World Wide Web*. ACM, 2004, pp. 40–52.

[31] K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL-injection Detection," in *Proceedings of the 2008 ACM Symposium on Applied Computing*. ACM, 2008, pp. 2153–2158.