Formulation of SQL Injection Vulnerability Detection as Grammar Reachability Problem

Kabir Umar
Faculty of Computer Science and Information Technology, Bayero University Kano, Gwarzo Road, Kano, Nigeria
ukabir.se@buk.edu.ng

Abu Bakar Sultan, Hazura Zulzalil, Novia Admodisastro, Mohd Taufik Abdullah
Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Selangor, Malaysia
{abakar, hazura, novia, taufik}@upm.edu.my

***Abstract* – Data dependency flow have been reformulated as Context Free Grammar (CFG) reachability problem, and the idea was explored in detection of some web vulnerabilities, particularly Cross Site Scripting (XSS) and Access Control. However, reformulation of SQL Injection Vulnerability (SQLIV) detection as grammar reachability problem has not been investigated. In this paper, concepts of data dependency flow was used to reformulate SQLIVs detection as a CFG reachability problem. The paper, consequently defines reachability analysis strategy for SQLIVs detection.**

Keywords: SQL Injection, Static Analysis, Vulnerabilities Detection, Web Application.

I. Introduction

For over a decade, exploitable software errors such as poor input validation, inclusion of user supplied data into generation of dynamic query, poor output sanitization, generous privilege, etc, constitute very serious security risks for web applications. Notable security reports such as OWASP Top 10 project (https://www.owasp.org/ index.php/ Category: OWASP_Top _Ten _Project) revealed that software errors leading to SQL Injection Vulnerability (SQLIV) are the most prevalent risk associated with web applications. Consequently, research communities have proposed several vulnerabilities detection techniques which modelled and investigated SQL injection problem using various underlying concepts such as parse tree analysis, control flow analysis, HTML page similarity analysis, graph of tokens analysis, etc. Unfortunately, SQLIVs detection has not been investigated using grammar reachability analysis.

Empirical evidence [1,2] has shown that reachability problems, such as graph nodes reachability and data dependency flow could be successfully reformulated as Context Free Grammar (CFG) non terminals reachability problems, and then be addressed using grammar reachability analysis. This idea was recently employed in addressing detection of some web vulnerabilities, particularly Cross Site Scripting (XSS) and Access Control [1,2]. This paper uses concepts of data dependency flow to reformulate the problem of SQL injection detection as a CFG reachability problem. Moreover, the paper defines grammar reachability analysis for detection of SQLIVs in web application. The remaining of this paper is organized as follows. Section II presents related works that addressed detection of vulnerabilities using grammar based analysis. Section III presents the proposed grammar reachability analysis for detection of SQLIVs. The section begins by introducing a running example web application that is used to support explanations in the paper. This is followed by discussion on the problem formulation. Extraction of grammar production rules from source code is also presented, and finally, the section defines the grammar reachability analysis strategy. Section IV presents possible applications and future work. Section V gives conclusion remark.

II. RELATED WORKS ON Grammar Based Analysis for Detection of Vulnerabilities

Not much work exist in addressing the problem of SQL injection vulnerabilities using grammar-based analysis, though, empirical evidences [1-5] reveal that context free grammar analysis has been successfully applied to improve static analysis process, as well as to address range of other problems. Wassermann & Su [3,4] proposed a combined static and dynamic analysis technique for prevention of SQL injection attacks based on syntactical isolation of user input within dynamically generated query. At static phase, the technique defines context free grammars for each query generated at hotspot. At dynamic phase, the technique uses context free grammar derivability to check syntactical confinement of input values within structure of dynamic query. Input values that violates this requirement are considered as evidence of possible attack, and the query is prevented from execution. It is worthy to note that, their proposed technique is for prevention of attacks, and does not address detection of vulnerability.

Minamide [1] proposed technique that approximates output of programs using language disjoint analysis. The technique traces sensitive sinks at which output operation is performed, and defines a context free grammar that approximates the output operation. The technique requires input of regular expression which defines unsafe input strings. The language of the context free grammar and the language of the regular expression are compared for disjointedness. If the two are disjoint, then the output generated by the sensitive sink operation is safe. This tech was used for detection of XSS (Cross-Site Scripting) vulnerabilities, and for validation of dynamically generated HTML pages. Sun et al., [2] proposed grammar based analysis for detection of access control vulnerabilities. Their technique model webpages as nodes, and defines context free grammar that represent possible HTML outputs of each node (The outcome produces set of webpages that are reachable from the node). The technique uses implicit information found inside each webpage to build per-role sitemaps, which also produces set of navigable pages. Finally, the technique compares the two sets to determine forced browsing.

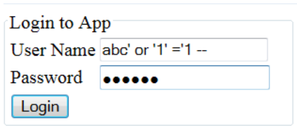
Reps [5] proposed approach that formulates shape-analysis problems as generalized graph reachability problems, and consequently used context free grammar reachability analysis to address it. Their technique was used for approximating possible "shapes" that can be taken by program's heap-allocated structures. The work of Christensen, Møller & Schwartzbach [6] proposed technique for evaluating possible values that can results from string expression. Their technique traces sensitive sink at which string operation is performed, and defines context-free grammar analysis to statically check syntax of dynamically generated expressions, such as SQL queries. The aforementioned works inspired our research in a number of ways. For instance, the work of Minamide [1], Reps [5] and Christensen et al. [6] inspired us to formulate the problem of establishing data dependency flow path as a graph reachability problem that can be addressed using context free grammar reachability analysis. In addition, we adapted part of the strategies use by [1] and [5] for extraction of grammar production rules from source code of web application.

III. PROPOSED GRAMMAR REACHABILITY ANALYSIS FOR DETECTION OF SQLIVS

This section presents principles supporting reformulation of SQLIVs detection as grammar reachability problem. The section also describes how CFG production rules are extracted from source code of web application, and then describes reachability analysis for detection of SQL injection vulnerable parameters. However, the section begins by introducing a running example web application and some important terminologies.

A Running Example Web Application

This section presents an example webpage which performs basic user authentication in a Java web application. The webpage is used as a running example to illustrate principles behind the problem reformulation. The example webpage contains form with two data input fields, namely: "username" and "Password". The form and source code for the webpage are shown in Fig. 1 (a) and (b) respectively. From the source code, it can be seen that data input field "username" is validated at line 13 using the validation function toSQL(N), whereas data input field "Password" is not validated. Thus "Password" field is vulnerable to SQL injection.


Login to App
User Name abc' or '1' ='1 --
Password ••••••

(b) Sample Login form (with user input)

```
1    protected void doPost(HttpServletRequest
2    request, HttpServletResponse response) {
3
4    String N =””;
5
6    String Q = null;
7    String R = null;
8    java.sql.Statement stat = null;
9    stmt = conn.createStatement();
10   java.sql.ResultSet S = null;
11   N = request.getParameter("username");
12   String P = request.getParameter("userpass");
13   R = toSQL(N); // validation for N (i.e.
     Username)
14   Q = "select * from userstbl where uname='"
15   + R + "' AND passwd ='" + P + "'";
16
17   S = Stmt.executeQuery(Q);
18    // exec qry at sensitive sink
19   }
```

(a) Code fragment from JSP servlet that receives the input data from form in (a), and perform basic User Authentication.

Fig. 1. Running Example Webpage

Some terminologies are explained with aid of the running example. These terminologies are used several times while explaining reachability and how SQL injection is modelled as a reachability problem.

i. Application's Entry Point (AEP): Statement at which input data gets into web application, e.g., lines 11 and 12.

ii. AEP Function (aep_fun): Functions that identify AEP statement, e.g., request.getParameter("username") of line 11.

iii. Sensitive Sink (SS): Statement at which dynamic query is executed is referred to as Sensitive Sink (SS), e.g., line 17 (Medeiros et al., 2014; Halfond & Orso, 2005; Yan et al., 2013).

iv. Sensitive Sink Function (ss_fun): Functions that identify Sensitive Sink statement. They invoke calls to database server operations, e.g., Stmt.executeQuery(Q) of line 17.

v. Query String: Defines SQL query to be executed, e.g., variable Q at line 14 defines the SQL query that is executed at line 17.

vi. Data Validation Statement: Refers to statement that contains validation or sanitization of input data, e.g., statement R = toSQL(N) of line 13

vii. Data Validation Function (val_fun): Functions that perform validation of data, e.g., the statement N = toSQL(N) of line 15 contains a call to validation function toSQL(N) which validates variable N, and return safe value. It validates the username input data.

B Formulation of Problem

The idea is to establish flow paths reachability from AEPs to SSs, and to analyze each established flow path for detection of SQLIVs. The flow paths establishment is inspired by concepts of data dependency flow graph which depicts "define-use" relationship between variables in source code, and can, thus, reveal paths along which variables pass data, for example, the path via which an AEP-parameter passes data to dynamic query at SS (i.e. an "AEP-to-SS" flow path). In a well secured application, data validation must be performed along such path. Obviously, when no data validation is performed along an "AEP-to-SS" flow path, then SQLIV is found.

```
9     stmt = conn.createStatement();
10    java.sql.ResultSet S = null;
11    N = request.getParameter("username");
12    String P = request.getParameter ("userpass");
13    R = toSQL(N);
14    Q = "select * from userstbl where uname='" +
      R + "' AND passwd ='" + P + "'";
17    S = Stmt.executeQuery(Q);
```

Fig. 2. Assignments and declarations statements of running example.

Consider the running example of Fig. 1, all statements which affect data dependencies along AEPs to SS paths are extracted and shown in Fig. 2, and the corresponding data dependency flow graph is shown in Fig. 3. It is worthwhile to note that in Java source code, only assignment statements and declaration statements are involved in data dependency flow. In the data dependency flow graph of Fig. 3, nodes represent variables. A "define-use" edge connects two nodes X and Y if variable represented by node X is used in the expression of variable represented by node Y.

The flow graph of Fig. 3 reveals the following information regarding the running example webpage.

i. There exist a data flow path "N − R − Q − S " which connects node N to node R to node Q and ends at node S. This path indicates flow of data from AEP parameter "username" (i.e., node N) into dynamic query execution at SS (i.e., node S). Observe that data validation is applied to variable N in the expression of variable at node R, therefore, "username" is secured.

ii. There exist another data flow path "P − Q − S " that begins from node P to node Q and ends at node S. This path indicates data flow of AEP parameter "userpass" (i.e., node P) into dynamic query execution at SS (i.e., node S). Observe that no data validation is applied along this path, therefore, "userpass" is vulnerable to SQL injection.

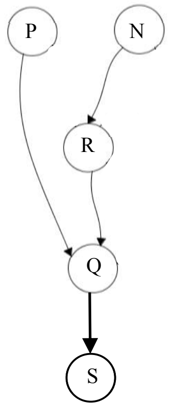


Fig. 3. Data Dependency Flow Graph for the running example

Establishing the above data flow paths can be considered as graph nodes reachability problem in which the target is to test reachability from one node (representing AEP statement) to another node (representing SS statement) [5]. This kind of problem can be modelled as context free grammar reachability problem and be solved using grammar based string analysis [1,5,7]. The strategy of our grammar based string analysis is to extract grammar production rules from declaration and assignment statements, such that the nonterminal symbol of the LHS (Left Hand Side) of each grammar production rule represents the variable of the LHS of the corresponding statement from which the grammar rule is extracted. The extracted grammar rules are converted to context free grammar production rules. Then, the CFG rules are used to test reachability from nonterminal symbol representing AEP statement to nonterminal symbol representing SS statement. However, in the extracted grammar, the rule extracted from a SS statement is considered as the start rule for tracking reachability.

Obviously, reachability in this context means existence of an AEP-to-SS data flow path. Where reachability is found, we analyze the corresponding productions sequence (i.e., derivations sequence) which establish the reachability for presence of data validation. Where data validation is found along productions sequence, then the associated AEP parameter is secured and not vulnerable. However, absence of data validation along such productions sequence means that the associated AEP parameter is not secured, and thus, SQLIV is found. Consequently, we propose reformulation of SQL injection vulnerability detection as grammar reachability problem in the following two steps.

i. Extraction of CFG production rules from all declaration and assignment statements of subject web application.

ii. Application of grammar reachability analysis on all "productions sequences" which establish reachability from nonterminal symbols representing SS statements to nonterminal symbols representing AEP statements.

The first step generates CFG production rules that serves as input to the second step which applies the proposed grammar reachability analysis strategy for detection of SQLIVs. The next section describes extraction of grammar production rules.

C Extraction of CFG Production Rules

Two types of grammar production rules are extracted from declaration and assignment statements. These are:

i. X-rules associated with type X set of nonterminals,

ii. Y-rules associated with type Y set of nonterminals

Type X nonterminals are of the form X_{ab}, and type Y nonterminals of the form Y_{ac}, where subscript "a" is line number of statement, "b" is a letter that identify type of statement ("S" for sensitive sink statement, "A" for AEP statement, "V" for data validation statement, or left blank for all other statements). Subscript "c" is a letter that identify sub-expression in statement ("a" for first sub-expression, "b" for second, "c" for third, and so on).

The X_{ab} nonterminal symbols represent variables, whereas the Y_{ac} nonterminal symbols are added to create grammar production rules for sub-expressions found in an X-rule. The X-rules produce combination of nonterminal and terminal symbols, whereas the Y-rules produce sequence of terminal symbols of sub-expressions. The set of alphabets for the terminal symbols of the extracted grammar is the same as the set of alphabets for the programming language of source code being analyzed. The extracted grammar has production rules of the form shown in Table 1. These format are adapted from format of grammar production rules extracted in [6].

TABLE 1. FORMAT OF EXTRACTED GRAMMAR PRODUCTION RULES

Example	Description
$X_{ib} \rightarrow X_j$	Unit rule, nonterminal produce single nonterminal
$X_{ib} \rightarrow \alpha X_j \beta \;\; where \;\; \alpha, \beta \in (\Sigma \cup N)^*$	Nonterminal produce combinations of terminals and nonterminals
$Y_{kc} \rightarrow \gamma \;\; where \; \gamma \in \Sigma^*$	Nonterminal produce sequence of terminal symbols
$X_{ib} \rightarrow \alpha$ **fun_name**$(X_j)\beta$	Nonterminal symbol produce function with one argument, e.g. **executeQuery**(X_p), **toSQL**(X_p)

Note that, in the table, X_{ib} , X_j and Y_{kc} denotes nonterminal symbols, the subscript *i. j. k.* represent statement's line numbers, the subscript b = letter S or A or V (as explained earlier), the subscript c is a letter that indicate sub-expressions' count, whereas the **fun_name** represents function names, denoting sensitive sink functions, AEP functions, and data validation functions.

The grammar rules extraction procedure is summarized in the following steps which are performed for each declaration and assignment statement in source code.

i. Add type X nonterminal symbol to represent LHS variable of statement. Append statement's line number as subscript to nonterminal symbol. Append subscript letter to indicate type of statement.

ii. Add an X-rule grammar rule for the above type X nonterminal symbol such that the nonterminal symbol produces what is on RHS (right hand side) of its corresponding statement.

iii. Replace each variable on RHS of the newly added grammar rule with corresponding type X nonterminal symbol.

iv. If the RHS of newly added rule contains sub-expressions between type X nonterminal symbols then replace each sub-expression with new type Y nonterminal symbol, and add a Y-rule for each of the type Y nonterminals such that the nonterminal produces the sub-expression. Append line number of corresponding statement as subscript to the type Y nonterminal. Append subscript letter that keep count of sub-expressions, such that letter "a" for first sub-expression, letter "b" for second sub-expression, and so on.

$X_{17S} \rightarrow$ **executeQuery**(X_{14})
$X_{14} \rightarrow Y_{14a}\, X_{13V}\, Y_{14b}\, X_{12A}\, Y_{14c}$
$Y_{14a} \rightarrow$ select * from userstbl where uname=' +
$Y_{14b} \rightarrow$ + "' AND passwd = '" +
$Y_{14c} \rightarrow$ + "'"
$X_{13V} \rightarrow$ **toSQL** (X_{11A})
$X_{11A} \rightarrow$ **request. getParameter**("username")
$X_{12A} \rightarrow$ **request. getParameter**("userpass")

Fig. 4. CFG rules extracted from the running example

v. If there is a sensitive sink function, an AEP function, or a data validation function on the RHS of newly added rule, then mark the function's name as special terminal symbol with bold font.

By applying the above grammar extraction steps to our running example, CFG rules listed in Fig. 4 are extracted.

D Grammar Reachability Analysis

In CFG, reachability exist between two nonterminal symbols if there are series of consecutive derivations (i.e., productions) that begins at the first nonterminal symbol and eventually produces sentential form containing the second nonterminal symbol [8 - 9]. The list of nonterminals on the LHS of production rules in a consecutive derivations is commonly referred to as "productions sequence" [10]. When a "productions sequence" establishes reachability then we refer to such as "reachability productions sequence". For the reformulation of SQLIVs detection as grammar reachability problem, we seek to analyze SS-to-AEP reachability for presence of data validation. Consequently, we define SS-to-AEP reachability as follows.

Definition 1: SS-to-AEP Reachability
Given a context free grammar $G = \{N, \Sigma,\ \Phi,\ S\}$ where the grammar rules are extracted from assignment and declaration statements of web application, and any two nonterminal symbols X_{iS} (representing SS statement at ith line number) and X_{jA} (representing AEP statement at jth line number). The nonterminal X_{jA} is said to be reachable from X_{iS} if there exist sequence of derivations that start with X_{iS} and eventually produces sentential form containing X_{jA}, mathematically expressed as follows.

$X_{iS} \overset{+}{\Rightarrow} \alpha X_{jA} \beta$ where $\alpha,\ \beta \in (\Sigma \cup N)^*$ and i, j are statement line numbers

Fig. 5. Definition of Reachability of CFG Nonterminals.

Using CFG terminologies, the reachability definition in Fig. 5 is read as "derivation of sentential form $\boldsymbol{\alpha X_{jA} \beta}$ starting from nonterminal symbol $\boldsymbol{X_{iS}}$", and can thus be represented in terms of the following "productions sequence" shown in Fig. 6.

$X_{iS}, \dots, X_{jA}$

Fig. 6. Productions Sequence

Once an SS-to-AEP reachability is established, and the corresponding productions sequence obtained, we analyze the grammar production rules along the productions sequence for detection of SQLIV. If data validation is applied on any nonterminal symbol along the productions sequence, then the associated AEP parameter is secure. If no data validation is applied along the productions sequence, then the associated AEP parameter is vulnerable to SQL injection. In view of this explanations we propose the following formal definition of SQLIV in terms of CFG reachability productions sequence.

Definition 2: SQL Injection Vulnerability (SQLIV)
Given a context free grammar $G = \{N, \Sigma,\ \Phi,\ S\}$ where the grammar rules are extracted from assignment and declaration statements of web application, any arbitrary AEP parameter denoted by "aep_name" is said to be vulnerable to SQL injection if there exist reachability from X_{iS} to X_{jA}, such that X_{iS} is a nonterminal symbol representing sensitive sink statement at line i, and X_{jA} is a nonterminal symbol representing AEP "aep_name" statement at line j, and no data validation is applied along the reachability productions sequence. Mathematically, the definition is expressed as in Fig. 7.

Given the SS-to-AEP Reachability:
$X_{iS} \overset{+}{\Rightarrow} \alpha X_{jA} \beta$ where $\alpha,\ \beta \in (\Sigma \cup N)^*$

An AEP parameter "aep_name" is vulnerable to SQL injection, if the following conditions apply:

i. $X_{iS} \overset{+}{\Rightarrow} \alpha X_{jA} \beta$ *corresponds to* $X_{iS}, X_k, \dots, X_l, X$

ii. $X_{iS} \rightarrow \mathbf{ss\_fun}(X_k) \in \Phi,\ \mathbf{ss\_fun} =$ Sensitive sink function name

iii. $X_{jA} \rightarrow \alpha\, \mathbf{aep\_fun}(\text{"aep\_name"})\beta \in \Phi,\ \mathbf{aep\_fun} =$ aep function name

iv. $X_{pV} \rightarrow \delta\, \mathbf{val\_fun}(X_{jA})\omega \notin \Phi,\ \mathbf{val\_fun} =$ data validation function name

v. $\{X_{pV}\} \cap \{X_{iS}, X_k, \dots, X_l, X_{jA}\} = \varepsilon$ *and* $\delta,\ \omega \in (\Sigma \cup N)^*$

Fig. 7. Definition of SQLIV

Considering the CFG extracted from source code of the running example (shown in Fig. 4), let us apply the above SQLIV definition and perform manual reachability analysis of the path from nonterminal symbol X_{17S} (representing sensitive sink) to nonterminal X_{12A} (representing AEP "userpass") as shown in Fig. 8. The reachability analysis involves testing for SQLIVs detection conditions. It can be seen from the figure that, all the five conditions are satisfied. This is because no data validation is applied to the nonterminal symbol X_{12A} along the productions sequence. Observer that the fourth condition is satisfied because there is NO data validation rule of the form $X_{pV} \rightarrow \delta\, \mathbf{val\_fun}(X_{12A})\omega$, which applies on X_{12A}, in the set of extracted grammar production rules Φ. The fifth condition is satisfied because the intersection of corresponding productions sequence $\{X_{17S}, X_{14}, X_{12A}\}$ and "applicable validation rule" is **empty**. Consequently, the AEP "userpass" is found to be vulnerable to SQL injection, and therefore, SQLIV is detected.

- Expression for the reachability to test is:
  $X_{17S} \overset{+}{\Rightarrow} \alpha X_{12A} \beta$
- We apply applicable derivations, and get the corresponding reachability productions sequence as:
  X_{17S}, X_{14}, X_{12A}
- We perform reachability analysis by testing SQLIV conditions. Note that ✓ means a condition is satisfied.

✓ i. $X_{17S} \overset{+}{\Rightarrow} \alpha X_{12A} \beta$ *corresponds to* $X_{17S}, X_{14}, X_{12A} \mid X_{17S}, X_{14}, X_{12A} \in \mathrm{N}$

✓ ii. $X_{17S} \rightarrow \textbf{ss\_fun}(X_{14}) \in \Phi,\ \textbf{ss\_fun} =$ Sensitive sink function

✓ iii. $X_{12A} \rightarrow \alpha\ \textbf{aep\_fun}(\text{"userpass"})\ \beta \in \Phi,\ \textbf{aep\_fun} =$ aep function

✓ iv. $X_{pV} \rightarrow \delta\ \textbf{val\_fun}(X_{12A})\omega \notin \Phi,\ \textbf{val\_fun} =$ datavalidation function

✓ v. $\{X_{pV}\} \cap \{X_{17S}, X_{14}, X_{12A}\} = \varepsilon$ *and* $\delta, \omega \in (\Sigma \cup \mathrm{N})^*$

Fig. 8. Analysis of SS to AEP “userpass” Reachability Productions Sequence

Similarly, by performing manual reachability analysis of the path from nonterminal symbol X_{17S} (representing sensitive sink) to nonterminal X_{11A} (representing AEP “username”), the reader should be able to see how proposed reachability analysis confirms that AEP parameter “username” is secured and not vulnerable to SQL injection.

IV FUTURE WORK

The above reformulation strategy is implemented into an Evolutionary static analysis method for detection and removal of SQL injection vulnerabilities, The method employs search strategies of variations and fitness evaluations to evolve “productions sequences” that establish SS-to-AEP reachability. Then, CFG reachability analysis (as described above) is applied on each optimal candidate for detection of SQLIV. Additional highlights on our research work can be found in [11, 12]. However, details of the search based method, EPSQLiFix is in the process of being published. Future work could include investigation of the search based grammar reachability analysis for detection of related vulnerabilities of web application, such as XSS.

V. CONCLUSION

This paper proposed reformulation of the problem of SQL Injection vulnerability detection as context free grammar reachability problems addressable using grammar reachability analysis. The paper defines strategy for extraction of context free grammar production rules from source code. Also, the paper defines grammar reachability analysis for SQL Injection vulnerability detection, and demonstrated how such analysis is performed with aid of a running example. In addition, the paper propose formal definition for SQL injection vulnerability in terms of grammar productions sequence. The reformulated SQL injection vulnerability detection strategy is currently implemented is an evolutionary static analysis method for detection and removal of vulnerabilities, named EPSQLiFix.

ACKNOWLEDGEMENT

We acknowledge that this research received support from the Fundamental Research Grant Scheme FRGS/1/2015/ICT01/UPM/02/12 awarded by Malaysian Ministry of Education to the Faculty of Computer Science and Information Technology at Universiti Putra Malaysia.

REFERENCES

[1] Minamide, Y. “Static approximation of dynamically generated web pages”. In Proceedings of the 14th International Conference on WWW, pp. 432-441, 2005.

[2] Sun, Fangqi, Liang Xu, and Zhendong Su. "Static Detection of Access Control Vulnerabilities in Web Applications." *USENIX Security Symposium*. 2011.

[3] Su, Zhendong, and Gary Wassermann. "The essence of command injection attacks in web applications." *ACM SIGPLAN Notices*. Vol. 41. No. 1. ACM, 2006.

[4] Wassermann, Gary, and Zhendong Su. "Sound and precise analysis of web applications for injection vulnerabilities." *ACM Sigplan Notices*. Vol. 42. No. 6. ACM, 2007.

[5] Reps, Thomas. "Shape analysis as a generalized path problem." *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 1995.

[6] Christensen, Aske Simon, Anders Møller, and Michael I. Schwartzbach. "Precise analysis of string expressions." *International Static Analysis Symposium*. Springer, Berlin, Heidelberg, 2003.

[7] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. "Introduction to automata theory, languages, and computation." *Acm Sigact News* 32.1 (2001): 60-65.

[8] Melski, David, and Thomas Reps. *Interconvertbility of set constraints and context-free language reachability*. Vol. 32. No. 12. ACM, 1997.

[9] Wikipedia. (2016d, January 11th). Chomsky Hierachy. [Online]. Retrieved from: https://en.wikipedia.org/wiki/Chomsky_hierarchy.

[10] Wikipedia. (2016g, January 18th). Context-Free Grammar. [Online]. Retrieved from: https://en.wikipedia.org/wiki/Context-free_grammar.

[11] Umar, Kabir, et al. "Prevention of attack on Islamic websites by fixing SQL injection vulnerabilities using co-evolutionary search approach." *Information and Communication Technology for The Muslim World (ICT4M), 2014 The 5th International Conference on*. IEEE, 2014.

[12] Umar, K., Sultan, A. B., Zulzalil, H., Admodisastro, N., & Abdullah, M. T. “On the automation of vulnerabilities fixing for web application”. In *Proceedings of The Ninth International Conference on Software Engineering Advances (ICSEA 2014)*, (pp. 221-226), Nice, France: IARIA, 2014, October 12-16.