Enhanced Pushdown Automaton based Static Analysis for Detection of SQL Injection Hotspots in Web Application

Kabir Umar¹,²*, Abu Bakar Md Sultan¹, Hazura Zulzalil¹, Novia Admodisastro¹, and Mohd Taufik Abdullah¹

¹Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang, Selangor, Malaysia
²Faculty of Computer Science and Information Technology, Bayero University Kano, Gwarzo Road, Kano, Nigeria

Abstract

SQL injection Hotspots (SQLiHs) are Application’s Entry Points (AEPs) through which SQL injection is possible, subject to the application’s internal sanitization or validation capabilities. Since not all AEPs are SQLiHs, one serious challenge during testing of very large web application for detection of SQL Injection Vulnerabilities (SQLIVs) is how to reliably decide which AEPs to consider for the test and which AEPs are unnecessary? In this paper, we propose a new Pushdown Automaton (PDA) based static analysis technique for detection of SQLiHs in web applications. The goal is to produce concrete information that can reliably and confidently guide both human tester/developer and SQLIVs detection tools/techniques as to which part of the source code to concentrate their efforts during detection and fixing of SQL injection flaws in an application. The proposed technique is an integral part of an on-going research on automated method for detection and removal of SQLIVs in web application. Experimental evaluation of the method is in progress. However, preliminary results show that the proposed technique is both feasible and effective.

Keywords: Context Free Grammar, Data Flow Path, Sensitive Sink, Vulnerabilities

1. Introduction

Detection of SQL injection Hotspots (SQLiHs) is means of recognizing Application’s Entry Points (AEPs) through which SQL injection is possible, subject to the application’s internal sanitization or validation capabilities. This is very important to both human tester/developer and SQL Injection Vulnerabilities (SQLIVs) detection tools/technique, because, if we can produce reliable information as to which AEPs can lead to SQL injections and which AEPs can never lead to SQL injections, then both human tester/developer and SQLIVs detection tools/techniques can be reliably and confidently guided as to which part of the source code to concentrate their efforts during detection and fixing of SQL injection flaws in an application.

Obviously, in the absence of such reliable information, the tester/developer or SQLIVs detection tool/technique may waste substantial amount of useful time and resources checking for vulnerabilities where they never exists. Although this problem is not an issue when testing a small application, however, it can turn into very serious concern as the size of application under test increases to several hundreds of thousands of LOCs. For example, if a developer is required to test large web application with hundreds of data entry fields, how does he/she reliably and confidently select which fields to consider during the security testing? Obviously, considering all fields may lead to waste of time and resources if not all the fields can lead to SQL injection flaws.

As part of an on-going research on automated method for detection and removal of SQLIVs in web application¹⁰,¹¹, we propose a new Pushdown Automaton (PDA) based static analysis technique that uses Context Free Grammar (CFG) analysis for detection of SQLiHs in web application.
applications. Our proposed technique provides reliable information to guide human tester/developers or SQLIVs detection tools/techniques in the scenario described above. The proposed technique is an integral part of the overall method. Experimental evaluation of the method is in progress. However, preliminary results show that the proposed technique is both feasible and effective.

The remaining of this paper is organized as follows. Sub Section 1.1 presents some terminologies used in describing the proposed technique. In Section 2 we present some existing work on Static Analysis for SQL injection flaws. Section 3 presents SQL Injection Hotspots in web application. Section 4 presents PDA based Static Analysis for Detection of SQLiH. The principles of the proposed technique is presented in sub Section 4.1. Thereafter, the proposed PDA is described in sub Section 4.2, while the system architecture is presented in sub section 4.3. Finally, conclusion and future work are presented in Section 5.

1.1 Important Terminologies

Some terminologies are used in describing the proposed technique. These terminologies are illustrated with the aid of Figure 1.

- **Application Entry Point**: Any location at which input data gets into an application is referred to as Application Entry Point (AEP). For example, lines 4 and 5 are AEPs in Figure 1.
- **Sensitive Sink**: The location at which dynamic query is executed inside web application is referred to as Sensitive Sink (SS). E.g., line 17 is a SS in Figure 1.
- **Target Operation**: To execute dynamic query at SS, a database server function is invoked with the “query string” as one of its arguments. For example, in line 17, the database server function “executeQuery()” is invoked with “DynSqlStr” as its argument. The function invoked to execute the query string is the Target Operation. In this example, the target operation is “executeQuery()”.
- **Target Operand**: This is the operand that is passed as argument to the target operation at SS. For example, in line 17, the string variable “DynSqlStr” is passed as argument to the target operation “executeQuery()”, therefore, in this example, the target operand is the string variable “DynSqlStr”.

![Login to App](image)

(a) Sample Login form (with user input)

```java
protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    String n = getParameter(request, "username");
    String p = getParameter(request, "userpass");
    String DynSqlStr = null;
    // no input validation is done. This is vulnerable
    DynSqlStr = "select * from userstbl where uname='" + n + "' AND passwd='" + p + "';"
    stmt = conn.createStatement();
    stmt.executeQuery(DynSqlStr);
    }
```

(b) Code fragment from JSP servlet that receives the input data from login form in (a), and use the data to generate and execute dynamic query

**Figure 1.** Sample login form (with user input).

2. Static Analysis for SQL Injection Flaws

Generally, static analysis is a process of reviewing and investigating code of an application without actually running the application. The goal of static analysis is usually to get concrete information about some attributes of code such as security flaws or to make changes to code such as vulnerability correction. The code being analyzed could be in form of source, binary, or intermediate representation such as AST. The process of static analysis has
been widely used in detection of wide range of web application vulnerabilities such as SQLIVs\textsuperscript{1,3,4}, XSS\textsuperscript{4}, Denial of Service\textsuperscript{5}, etc. Unfortunately, the existing literature is short of techniques that specifically detects SQLiHs in web applications.

Several approaches for performing static analysis exist in the literature. Some static analysis tools perform semantic analysis over nodes of AST representation of the application being analyzed. For example, WAP (Web Application Protector) uses AST walkers generated using ANTLR tool to analyze nodes of application's AST for detection of vulnerabilities\textsuperscript{1}. Other tools perform data flow analysis to collect information about how data flows within an application that is in a static state\textsuperscript{2}. Data flow analysis in which flows of “tainted” data is analyzed is commonly known as taint analysis\textsuperscript{2}. In taint analysis, data which enters the application is marked as “tainted”. The path followed by the tainted data is traced to see if it reaches Sensitive Sink (SS) or not. Presence of vulnerability is reported if tainted data is found to reach SS\textsuperscript{1,3,4}. Pixy\textsuperscript{5} extends taint analysis with alias analysis to trace tainted data through multiple variables that passes the same value. Safe PHP applies taint analysis to detect vulnerabilities leading to denial of services and unauthorized database operations\textsuperscript{5}.

Common problem of most existing static analysis techniques discussed above is false positives reporting\textsuperscript{1,2}. One reason for this is because taint analysis techniques concentrate on tracing data flow in variables without taking the grammatical connections between statements into account. WAP\textsuperscript{1} addressed false positives problem by performing data mining to predict false positive within results of taint analysis. Although their approach was effective at reducing false positive in the final result, but it did not improve the actual taint analysis process in any way, the data mining only predicts false positives in the results of static analysis. Our technique aims at improving static taint analysis process by integrating PDA based Context Free Grammar (CFG) analysis into the process. The following section presents explanation of SQL injection hotspots with respect to web applications.

3. SQL Injection Hotspots in Web Application

In web application, dynamic queries are generated at runtime by combining SQL strings with values collected from external source such as user input or from data stores such as database table or cookie stored data\textsuperscript{6,7}. Figure 1 (a) shows a sample login form that provides two input data entry fields (“User Name” and “Password” fields) for collecting data from external source, i.e., the “user” in this context.

Code fragment of JSP servlet that receives the input data from the two fields is shown in Figure 1 (b). Typically, the input data collected via these fields are stored into variables as in lines 4 and 5 of Figure 1. (b). Thereafter, the variables are used in generation of dynamic query as in lines 13 and 14. Finally, the dynamically generated query is executed by database server as in line 17. Careful study of data flows within source code such as the fragment of Figure 1 (b) leads us to the following important assumption:

An input data entry field, such as “User Name” and “Password” fields of Figure 1, is classified as SQL Injection Hotspot (SQLiH) if any one of the following two conditions hold:-

- Input data collected via the field is directly used in construction and execution of dynamically generated SQL query.
- Input data collected via the field is stored into an input data variable, and the variable is eventually used in construction and execution of dynamically generated SQL query.

Therefore, SQLiHs are AEPs through which external data comes into the application and get channeled to dynamic queries at Sensitive Sink (SS). Potentially, all SQLiHs are susceptible to SQL injection. When proper sanitization or validation is done on the data coming into an application via a hotspot, then the hotspot is secured. However, when no proper sanitization or validation is done, the hotspot turns to a vulnerability through which the application can be exploited. Input data collected via SQLiH need to be sanitized or validated by the application prior to inclusion in generation of dynamic query (lines 13 and 14) and execution (line 17). Failure to sanitize or validate input data, collected from a SQLiH into dynamic query, is the root cause of SQL injection vulnerabilities. The code fragment of Figure 1 (b) does not perform any sanitization or validation of hotspot-collected input data, and is therefore, vulnerable to SQL injection. To confirm presence of SQLiH we must establish data flow paths from an AEP to SS. This can be achieved by means of Data Flow Graph (DFG) which can show data dependencies along all paths that might be traversed from AEP to SS as shown in Figure 2.
Enhanced Pushdown Automaton based Static Analysis for Detection of SQL Injection Hotspots in Web Application

Figure 2. Data flow graph for AEP-to-SS data flowpaths.

The DFG in Figure 2 illustrates the data flow paths from AEP “username” to the SS at line 17 in sample Login form example of Figure 1. The illustration establishes data flow paths from “Node AEP” to “Node A” to “Node D” and ends in “Node SS”. These data flow paths are shown with red coloured dashed-arrows labeled “username” in the figure. Thus, we have established that AEP “username” is a SQLiH. The idea behind our proposed technique emerges largely from close study of DFG and PDA. Next section discusses our technique in details.

4. PDA based Static Analysis for Detection of SQL Injection Hotspot

The proposed technique is for establishing concrete data flow paths that exist between AEPs to SSs. Each occurrence of such path translates into a possible SQLiH. In essence, our approach is a way of using PDA based Context Free Grammar (CFG) analysis to establish concrete evidence of possible AEPs via which input data is channeled to dynamic queries.

Our static analysis approach uses CFG analysis, hence, the approach is both syntax and semantic focused. This is a big improvement over most existence static analysis techniques that are mainly semantic focused. We use computational power of PDA to trace data source of input variables found in dynamic queries to an AEP, and consequently identifies all SQLiH in a very effective manner. In the following sub sections, we present the principles of the technique, followed by description of the proposed PDA for SQLiH detection as well as its system architecture.

4.1 Principles

We use the sample login form and code fragment of Figure 1 to describe principles of our technique. As explained in previous section, the sample login form contains two SQL injection hotspots, namely “User name” and “Password” fields. Therefore, any input data collected through these fields will eventually be used in construction and execution of dynamic query that performs user authentication at lines 13 to 17 of Figure 1. Out static analysis is based on the assumptions stated in Section 3.

Hence, for all possible dynamic queries that an application can generate and execute, if we can statically analyze the source code of the application and detect all statements that are involved in generation of each dynamic query, and identify all input data variables that will eventually be in each dynamic query, then it would be possible to trace the primary data source for each identified input data variable. Consequently, where the primary data source is traced to an AEP, then such AEP can easily be classified as a SQLiH. We propose formal definition of SQLiH as follows:

Figure 3. States diagram of proposed SQLiH recognizer PDA.

Let dynamically ‘generated query string’ be denoted as $Q_{\text{dyn}}$, let ‘input data variables’ found (i.e., used) in $Q_{\text{dyn}}$
be denoted as $V_1, V_2, V_3, \ldots, V_n$ ($n \geq 1$), and let there exist any arbitrary AEP denoted as $X_{AEP}$.

**Definition**

Any arbitrary AEP such as $X$ is classified as a SQLiH if and only if there exist at least one data flow paths from $X_{AEP}$ to $Q_{dyn}$ via any $V_i \in \{V_1, V_2, V_3, \ldots, V_n\}$.

The above data flow path can be traced using DFG such as Figure 2. However, careful study of the DFG in Figure 2, lead us to discovery of another approach for establishing $X_{AEP}$ to $Q_{dyn}$ (i.e., AEP to SS) data flow paths using grammar analysis strategies. Instead of tracing the data flow path from an AEP down to SS, our approach seek to perform the trace in a reverse way from SS up to AEP by applying the following procedure:

- In the application under analysis, consider every declaration and every assignment statement as a grammar production rule.
- Extracts all the input data variables $s$ in the operand statement or expression. For each extracted $V_i$, find the sequence of production rules that produces, directly or indirectly, until you reach an AEP or no more production rules are found.
- If $X_{AEP} \Rightarrow + V_i$ then $X_{AEP}$ is a SQLiH else it is not.

As can be seen from the above procedure, our static analysis approach takes into account both the syntax, semantics and context of relevant application’s statements during the analysis process. This is a significant improvement over most existing static analysis approaches which do not take all the three into account.

### 4.2 SQL Injection Hotspot Recognizer PDA

Pushdown Automaton (PDA), also known as Mealy machine, is an abstract machine that defines model of computation involving change of states, i.e., state transitions, based on three parameters namely, the current state, input alphabet, and topmost element of a stack. During each state transition, the machine can perform some actions which may change the topmost element in the stack. In addition, a state transition can be triggered by change in topmost stack element even when no input alphabet is available.

We designed an enhanced Pushdown Automaton (PDA) to perform the procedure described in the principles above. The state diagram for the proposed PDA is shown in Figure 3. The enhanced PDA consists of two stacks, namely “Primary stack” and “Auxiliary stack”. The states of the PDA are derived from nodes of DFG of Figure 2. Each of the PDA states corresponds to a statement along the SS-to-AEP dataflow path. The PDA operation begins at SS with push of “operand” into the primary stack. Tokens in primary stack top element are analyzed to decide state transition and next stack push operation. This is achieved by determining tokens that exist on right side of grammar production rules (i.e., statements). These tokens are considered as Non terminal symbols of our CFG rules. The auxiliary stack is used for temporary storage of “statements so far analyzed”.

In addition, the enhanced PDA uses look-ahead strategies to resolve any encountered non deterministic state transition and avoid the need for back tracking. Furthermore, when the PDA encounters multiple variables, e.g., $V_1, V_2, V_3, \ldots, V_n$ ($n \geq 1$), each of which must be traced separately and independently, our approach creates multiple instances of the PDA, one instance for each variable. This allows the analysis to scale very well.

### 4.3 Architecture of the Proposed System

In web application, SQL injection Hotspot (SQLiH) are input data entry fields (i.e., AEPs) through which external data comes into the application and get transmitted into dynamic queries at sensitive sinks (SS). SQLiH are potentially susceptible to SQL injection. The proposed technique is about establishing concrete data flow paths that exist between AEPs to SSs. Each occurrence of such path translates into a possible SQLiH.

The proposed technique performs lexical and parse analysis of the application’s source code to recognize individual statements and their tokens. In addition, at each SS statement, the technique employ PDA based grammar rules replacement strategies to establish existence of possible SS-to-AEPs data flow path. The technique consists of the following steps:

#### 4.3.1 Lexical and Parse Analysis

The technique analyzes application’s source code to generate tokens and recognize statements. During this process, symbol tables are produced for reference at later stages. The symbol tables include: 1. List of string variables declarations, 2. List of all string variables assignment statements, 3. List of recordset/sql_statement variables...
declarations, 4. List of all recordset/sql_statement variables assignment statements.

4.3.2 PDA based Analysis
The technique uses a two-stack n-instance PDA to perform grammar rules replacements analysis. Starting with target operation at SS, and guided by information from Symbols tables, the PDA analysis generates CFG production rules that can trace SS to AEP paths.

4.3.3 Reporting
The system generates and reports static analysis feedback. These consist of list of detected SQLiH, validated and un validated SQLiHs. For each SQLiH, the system generated a CFG rules that traces SS-to-AEP data flow paths. This report is generated from contents of auxiliary stack of the PDA. The system architecture of the proposed technique is shown in Figure 4.

5. Conclusion and Future Work
Having concrete information about which AEPs are hotspots for SQL injections can reliably and confidently guide both human tester/developer and SQLIVs detection tools/techniques as to what part of source code to concentrate on during testing of web application for detection of SQLIVs. In the absence of such reliable information, substantial amount of useful time and resources may be wasted by checking for vulnerabilities where they never exist. Although this problem is not an issue when testing small application, however, it can turns into very serious concern as the size of application under test increases to several hundreds of thousands of LOCs.

In this paper, we propose a new Pushdown Automaton (PDA) based static analysis technique for detection of SQLiHs in web applications. The PDA uses context free grammar rules replacement strategies to establish presence of SS-to-AEP data flow paths. The proposed technique is an integral part of an on-going research on automated method for detection and removal of SQLIVs in web application. Experimental evaluation of the method is in progress. We hope to produce separate experimental results for the proposed technique in due course. However, preliminary results show that the proposed technique is both feasible and effective.

6. 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.

7. References