SQLUnitGen: Test Case Generation for SQL Injection Detection

Yonghee Shin, Laurie Williams, Tao Xie

North Carolina State University

Abstract

More than half of all of the vulnerabilities reported can be classified as input manipulation, such as SQL injection, cross site scripting, and buffer overflows. Increasingly, automated static analysis tools are being used to identify input manipulation vulnerabilities. However, these tools cannot detect the presence or the effectiveness of black or white list input filters and, therefore, may have a high false positive rate. Our research objective is to facilitate the identification of true input manipulation vulnerabilities via the combination of static analysis, runtime detection, and automatic testing. We propose an approach for SQL injection vulnerability detection, automated by a prototype tool SQLUnitGen. We performed case studies on two small web applications for the evaluation of our approach compared to static analysis for identifying true SQL injection vulnerabilities. In our case study, SQLUnitGen had no false positives, but had a small number of false negatives while the static analysis tool had a false positive for every vulnerability that was actually protected by a white or black list. Future work will focus on removing false negatives from SQLUnitGen and at generalizing the approach for other types of input manipulation vulnerabilities.

1 Introduction

More than half of all of the vulnerabilities[1] reported in 2003-4 [28] were input manipulation vulnerabilities. Some examples of input manipulation include SQL injection, cross site scripting (XSS), and buffer overflows, according to the Open Source Vulnerability Database[2] classification. Input manipulation vulnerabilities exploit the fact that dynamic operations, such as SQL queries, can be constructed with user input as variables and that these dynamic operations are not always safe. For example, if an attacker enters SQL commands in a user input field, such as user name field, the resulting dynamically-generated SQL query may be altered from its intended function in the application and may enable the attacker to perform an unauthorized task.

By limiting user input such that only well-formed strings are accepted into the application, input manipulation vulnerabilities can be reduced. To do this, applications can filter user input via white lists[3], black lists[4], or a combination of the two, prior to allowing the input to reach the logic of the application. Validating input against a white list filter has been shown to be more feasible than matching against a potentially infinite black list [13, 15].

One means of detecting input manipulation vulnerabilities is the use of automated static analysis tools [6, 7]. However, these tools cannot detect the presence or the effectiveness of black or white list input filters. As a result, static analysis tools may have a high false positive rate when reporting input manipulation vulnerabilities in applications with effective filters. Alternatively, SQL injection attacks (SQLIA) can be automatically detected at runtime and prevent malicious SQL queries from being executed instead of rely-

[1] A vulnerability is a security flaw in the system that represents a valid way for an adversary to realize an adversary's goals [26].

[2] http://www.osvdb.com/

[3] A white list represents valid input as specified by requirements for a software system [13].

[4] A black list represents any input not defined as valid by the system requirements [13].

ing on input validation functions in the program [12, 25, 27] However, runtime detection does not provide information that can be used to fix the vulnerable code in the early development phase.

Our research objective is to facilitate the identification of true input manipulation vulnerabilities via the combination of static analysis, runtime detection, and automatic testing. Specifically, this paper reports our first step in this research objective, a prototype tool SQLUnitGen v0.5 that can be used to identify SQL injection vulnerabilities. As we refine SQLUnitGen, we expect that the principles and techniques embodied in this tool can be expanded for the identification of other types of input manipulation vulnerabilities.

To measure the effectiveness of SQLUnitGen v0.5, we performed case studies on two small web applications with the tool. We examined the ability of SQLUnitGen to detect SQLIAs for the applications with differing levels of input filtering. We compare these results with the vulnerability detection of FindBugs[5], a static analysis tool.

The rest of this paper is organized as follows. Section 2 provides background and Section 3 describes related work. Section 4 describes our approach. Section 5 describes our case studies and evaluation results. Section 6 concludes and discusses the future work.

2 Background

This section describes SQL Injection attacks with an example and describes the two tools, AMNESIA [12] and JCrasher [11], that SQLUnitGen is based on.

2.1 SQL Injection Attacks

Through a SQL query, a program can add, modify, or retrieve data in a database. SQL injection enables attackers to access, modify, or delete critical information in a database without proper authorization. Via SQL injection, attackers can also execute arbitrary commands with high system privilege in the worst case [2]. SQL injection has recently been one of the top issues in software security [1].

In many cases, SQL queries are dynamically constructed via user input. Despite there being several safer ways to make SQL queries in systems such as using Java's `PreparedStatement`, queries are often dynamically generated in string concatenations, an unsafe and poor programming practice. For example, Figure 1 shows a sample program including a SQL query to authenticate a user via id and password. The query is dynamically created via the program statement in bold. In the query in Figure 1, `id` and `password` are obtained via user input.

```
public boolean isRegistered(String id,
                            String password) {
 String driver = "com.mysql.jdbc.Driver";
 String to = "jdbc:mysql://cc.com/credit";
 Class.forName(driver).newInstance();
 Connection dbConn =
       DriverManager.getConnection(to);
String sqlQuery =
      "SELECT userinfo FROM users
       WHERE id = '" + id + "'
       AND password = '" + password + "'";
Statement stmt =
       dbConn.createStatement();
 ResultSet rs =
       stmt.executeQuery(sqlQuery);
 if(rs != null) return true;
 else return false;
}
```

Figure 1: An example of SQL query.

A SQL injection attack occurs when an input from a user includes SQL keywords so that the dynamically-generated SQL query changes the intended function of the SQL query in the application. In the previous example, an attacker can enter the following input through the user interface for the values of `id` and `password`:

```
Username: ' OR '1' = '1
Password: ' OR '1' = '1
```

Which would generate the following query:

```
SELECT userinfo FROM users
WHERE id = '1' OR '1' = '1'
AND password = '1' OR '1' = '1';
```

Because the given input makes the WHERE clause in the SQL statement always true (a tautology), the database returns all the user information in the table. Therefore, the malicious user has been authenticated without a valid login id and password. The use of tautology is a well-known SQL attack [2, 12, 25]. However, there are other types of SQLIAs using multiple SQL statements or stored procedures. SQL clauses such as "UNION SELECT", "ORDER BY", and "HAVING" are sometimes used to infer database

[5] http://findbugs.sourceforge.net/

structure. The attackers also can infer database structure by exploit error messages from SQL command failure [2, 21] or simply by trial and error [20].

2.2 AMNESIA

AMNESIA[12] is a runtime SQLIA detection tool. AMNESIA consists of two parts; static analysis and dynamic detection. During static analysis, AMNESIA identifies *hotspots* where a hotspot is defined as "points in the application code that issue SQL queries to the underlying database." [12]. Then, AMNESIA builds a model of SQL queries that could be generated by an application for each hotspot. At runtime, AMNESIA checks the dynamically-generated queries against the statically-built query model. If they do not match, the AMNESIA runtime monitor returns an error and prevents the SQL query from being executed. Otherwise, the query is sent to the database server.

The AMNESIA SQL query model is constructed based on Java String Analyzer (JSA) that uses a static string analysis technique [9]. JSA statically analyzes string data flow in Java programs and converts the flow graph to an approximated regular expression that can be easily converted into a finite state automaton. Because the approximation is conservative, the resulting automaton represents all the possible strings that can be represented at a particular location in a program but also could include some impossible strings.

AMNESIA converts the character-level automata generated by JSA into SQL query automata, which groups a SQL keyword into a transition. The transitions in the automata consist of SQL keywords, operators, constants, and a special keyword representing SQL query variables such as `id` in our example. Figure 2 shows the SQL query model built from the example SQL query in Figure 1. In Figure 2, β indicates a SQL query variable that holds user input.

Our approach modifies SQL query models generated by AMNESIA to trace the user input that reaches a SQL query and to generate attack input for the test cases. AMNESIA is also used as a test oracle to detect the SQLIA during test execution. A test oracle is a program or function that determines if a test execution passed or failed.

2.3 JCrasher

JCrasher automatically generates test cases for the methods in a class with predefined input values. For example, for integer types, JCrasher uses 1, 0, and -1 as input values. To generate test cases, JCrasher probes a type space, mapping a type to a set of pre-defined values or to methods returning the type, and constructs a parameter graph that represents possible combination of input values for parameters.

The reasons we chose JCrasher among other tools are as follows. JCrasher generates concrete input values for string types, which is crucial for our approach. Secondly, JCrasher generates JUnit test cases written in Java language. Thirdly, because its source code is available we could modify JCrasher for our purpose. Finally, even though JCrasher was not designed for high test coverage, our experience showed that the test coverage provided by JCrasher was sufficient for our initial prototype.

3 Related Work

There are existing tools and techniques that can be used in development and testing time to detect or prevent input manipulation vulnerabilities and to improve programs so that input manipulation vulnerabilities can be reduced. This section discusses those techniques and compares with our approach.

3.1 Manual Approaches

This section describes manual approaches to detect and prevent input manipulation vulnerabilities.

3.1.1. Defensive programming Many input manipulation attacks can be prevented, by implementing the application in a way that user input cannot contain malicious characters or keywords. Programmers can implement their own input filters by using white lists or black lists. Our approach helps to improve user input validation by providing information on vulnerable method arguments and malicious input. Programmers can

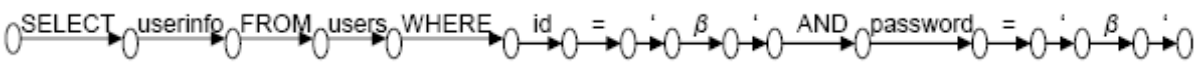


Figure 2: SQL query model.

also use existing safe APIs that prevent malicious input by strong type checking or converting malicious input into safer input. SQL DOM [22], Safe Query Objects [10], `PreparedStatement` in the JDBC API, and special APIs provided by DBMSs are in this category.

However, some of these approaches require programmers to learn the usage of APIs. Because learning new APIs takes time, programmers tend to use the APIs they know when the time is limited or to reuse the existing source code. Additionally, legacy code can contain SQL injection vulnerabilities. Furthermore, improper usage of `PreparedStatement` still can allow SQLIAs. Table names and column names cannot be used in a safe way even with `PreparedStatement`. Therefore, when a table name or a column name is used as a variable, it should be validated in the application [19].

3.1.2 Code review Code review is known to be effective in detecting bugs with low cost [5]. However, code review is a time consuming task compared to automated static analysis [7] and may be skipped by software development teams rushing to ship an application. In addition, the reviewer must have deep knowledge about how SQLIAs work. A strength of our approach is that it provides fast and early detection of vulnerabilities by non-security experts.

3.1.3 Manual penetration testing **"**Penetration testing is security testing in which evaluators attempt to circumvent the security features of a system based on their understanding of the system design and implementation" [6]. Penetration testing is usually performed at the end of development life cycle within a limited amount of time [3]. Therefore, the cost of removing the vulnerabilities found during penetration testing is very expensive. Penetration testing is usually performed in a black-box approach. Thus, the testing result does not directly inform the vulnerable location in the application. On the other hand, our approach tests applications during the unit testing and integration testing periods. Therefore, SQL injection vulnerabilities can be detected and fixed earlier than with penetration testing. In addition, our approach is a white-box approach and provides test results in a way programmers can easily identify the vulnerable location in the application.

[6] http://www.atis.org/tg2k/_penetration_testing.html

3.2 Automated Approaches

This section describes automated approaches to detect and prevent input manipulation vulnerabilities.

3.2.1 Static analysis FindBugs [14] is a static analysis tool that detects various bugs in Java programs, including SQLIAs. FindBugs gives a warning when a SQL query is constructed from variables instead of purely constant values. However, FindBugs does not assess whether the input was validated properly before the input is used in a SQL query or not and, therefore, may generate many false positives. On the other hand, our approach provides more precise information about SQLIAs based on test execution.

3.2.2 Web vulnerability scanning Web vulnerability scanners crawl and scan for web vulnerabilities by using software agents. These tools perform attacks against web applications and detect vulnerabilities by observing their behavior to the attacks [4, 18]. WAVES [16], SecuBat [18], AppScan, ScanDo, and WebInspect [4] are in this category. However, without exact knowledge about the internal structure of applications, it is difficult to generate precise attack input that can reveal input manipulation vulnerabilities. On the other hand, our approach uses more precise attack input based on static analysis on application source code and identifies proper attack input for method arguments.

4 Proposed Approach

User input might take a circuitous path from the user interface through one or more methods that may or may not be input filter methods, and ultimately to the SQL command to be executed. Our approach traces the flow of the input values that are used for a SQL query by using the AMNESIA SQL query model [12] and string argument instrumentation. Based on the input flow analysis, we generate test attack input for the method arguments used to construct a SQL query. We generate test cases with an existing JUnit test case generation tool, JCrasher, and modify the test input with attack input. To help programmers to easily identify vulnerable locations in the program, our approach generates a colored call graph indicating secure and vulnerable methods.

This section provides an overview of our approach that consists of three phases and describes each phase in the following sections.

4.1 Overview

Our research objective is to facilitate the identification of true input manipulation vulnerabilities via the combination of static analysis, runtime detection, and automated testing. Our objective is based on the overarching goal of software security in which security is an integral part of the development process [23].

Our approach involves three phases. In the first phase, test cases whose execution reaches SQL query statements are generated. We call these test cases *hotspot-reaching test cases*. During the second phase, the generated test cases are refined so that the input values of test cases are replaced with attack input. The attack input for method arguments is determined from an *augmented SQL query model* which has additional input flow information than a standard SQL query model. In the third phase, test cases are executed to detect vulnerabilities and test result summaries are generated. Programmers can then use the test result summaries to improve the program. Figure 3 shows the overview of test case generation process. The following sections describe each phase in detail.

4.2 Phase 1: Generate Hotspot-reaching Test Cases

Generating hotspot-reaching test cases is performed in two steps. At first, initial test cases are generated by JCrasher [11]. Then, hotspot-reaching test cases are collected from the initial test cases. To collect hotspot-reaching test cases, SQLUnitGen finds a hotspot from the Java application's byte code[7] and instruments (modifies) the byte code so that an exception is raised right before the hotspot is executed using BCEL (Byte Code Engineering Library)[8]. BCEL provides the APIs to analyze byte code and change programs directly on the byte code level instead of modifying the source code. Therefore, SQLUnitGen can search the method signature defined as hotspots using BCEL. Figure 4 shows an example of the instrumented code. For ease of reading, we show the instrumentation on the source code level, instead of byte code level. In this example, right before a hotspot `executeQuery` is executed, `HotspotException` is raised.

```
public boolean isRegistered(String id,
                        String password) {
 …
 throw new HotspotException();
 ResultSet rs =
      stmt.executeQuery(sqlQuery);
 if(rs != null) return true;
 else return false;
}
```

Figure 4: Hotspot instrumentation.

SQLUnitGen collects the test cases that raise the instrumented exception as hotspot-reaching test cases. The execution of these test cases is guaranteed to reach hotspots. Therefore, if the execution of these test cases with malicious/attack input does not reach a hotspot, the program has effectively blocked the malicious input. In Phase 2, these test cases are modified to include attack input.

4.3 Phase 2: Generate Attack Test Case

For attack test cases, we need to generate attack input so that the attack input does not cause SQL syntax or semantic errors unnecessarily. For example, a column in a database table with character data type must use single quotation marks properly so that a SQLIA will not generate a SQL syntax error. We also need to identify the flow of user input from methods to hotspots to identify which arguments of which methods must have the generated attack input. For these purposes, we use a SQL query model and string argument instrumentation, as described in Section 4.3.1 and Section 4.3.2.

[7] Java byte code is the code the Java compiler produces from Java source code. Java byte code is interpreted (executed) by a Java Virtual Machine (JVM).
[8] http://jakarta.apache.org/bcel/

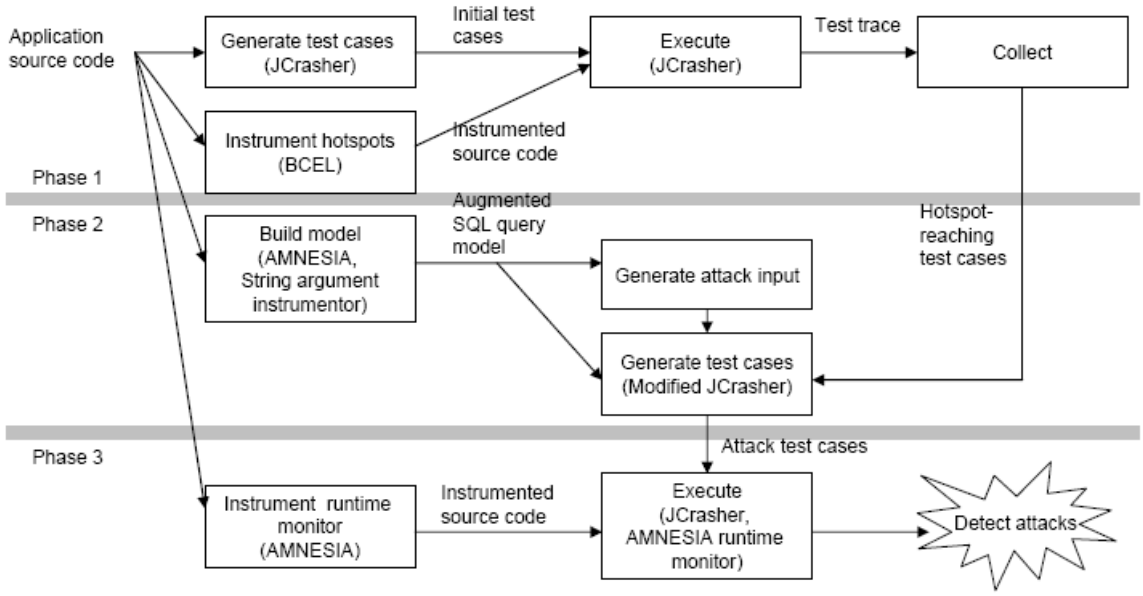


Figure 3: Test case generation process.

4.3.1 Building augmented SQL query model
To trace the flow of input values, SQLUnitGen instruments the byte code so that, for each string type argument, its method name and argument index are added before and after the string argument value. Figure 5 shows an example of instrumentation. For ease of reading, we show the instrumentation on the source code level, instead of byte code level. The instrumentation is in bold. In this figure, the argument `id` and `password` are string types. Therefore, the argument `id` is tagged with the method name `isRegistered` and argument index 0 and the argument `password` is tagged with the method name `isRegistered` and argument index 1.

The SQL query model built on the instrumented byte code includes the tagged information as if it is a part of the SQL query. We call the SQL query model with tagged information the *augmented SQL query model*. When the value of a variable cannot be determined in the application because it comes from user input, it is represented as a special keyword, β, in the SQL query model. The resulting augmented SQL query model is shown in Figure 6. In Figure 6, `id` and `password` are represented in β, which represents user input.

```
public boolean isRegistered(String id,
                            String password) {
  id = "[isRegistered-0]" + id +
       "[isRegistered-0]";
  password = "[isRegistered-1]" +
             password +
             "[isRegistered-1]";
       …
}
```

Figure 5: String argument instrumentation.

When an argument is passed through a chain of method calls, the tag includes all the methods involved in the method call chain. For example, if method `isRegistered` is called by method `Login` as Figure 7, the beginning tag of variable `id` becomes `[isRegistered-0][Login-0]` and the ending tag of variable `id` becomes `[Login-0][isRegistered-0]` as a result of

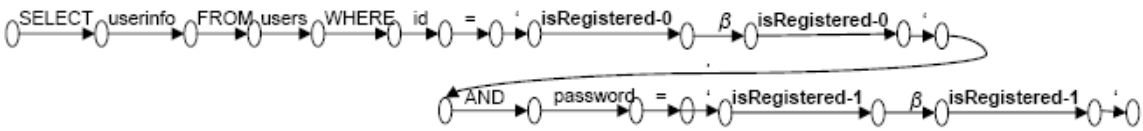


Figure 6: Augmented SQL query model.

string analysis.

```
public boolean Login(String id,
                     String password) {
   boolean result = isRegistered(id,
                                 password);
       …
}
```

Figure 7: Nested method call.

Figure 8 shows the augmented SQL query model for the program segment in Figure 7. Figure 8 shows only a part of the augmented SQL query model for the variable `id`. From this augmented SQL query, we can trace the flow of initial input to a SQL query and identify method arguments that are used to construct a SQL query.

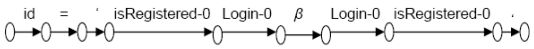


Figure 8:Augmented SQL query model for nested method calls.

4.3.2 Generating attack input For the successful attack, attack input should not cause SQL exceptions including SQL syntax errors or SQL semantic errors unnecessarily. SQL semantic errors occur when the attack input includes incorrect table names or column names. SQL syntax error occurs when the dynamically constructed SQL query from the attack input results in a syntactically illegal form. For example, if we use `"1' OR '1'='1"` as attack input for an integer type column of a database table, the query will generate a syntax error. In SQLUnitGen v0.5, we reduce syntax errors via the syntactical information we can obtain from the SQL query model.

However, some syntax errors are unavoidable. For example, when attack input includes multiple SQL statements, if a DBMS does not allow multiple SQL statements in a query execution, the DBMS generates a syntax error. When we count successful SQLIAs from test cases, we only count the test cases that were successful in SQL injection without causing SQL syntax errors.

As attack input, SQLUnitGen v0.5 includes nine attack patterns gathered from various resources [2, 17, 19, 21]. They are only a small subset of possible attack patterns. The nine attack patterns are described in Section 4.5 in detail. We will include more attack patterns in the future releases

4.3.3 Generating attack test cases From the previous two steps, we obtain the attack input for method arguments. To generate attack test cases with SQLUnitGen, we modified JCrasher so that JCrasher changes test input of hotspot-reaching test cases to include the attack input identified in the previous steps for corresponding method arguments. Figure 9 shows a test case for method `isRegistered` generated by JCrasher during the first phase. Figure 10 shows a modified test case containing attack input during the second phase. The test case in Figure 10 tests if the variable `id` in the example in Figure 1 is properly validated or not.

```
public void test0() throws Throwable {
      java.lang.String s4 = "normal";
      java.lang.String s5 = "normal";
      SampleApp s2 = new SampleApp();
      boolean result =
                s2.isRegistered(s4, s5);
}
```

Figure 9: Test case before modification.

```
public void test0() throws Throwable {
     java.lang.String s4 = "1' OR '1'='1";
     java.lang.String s5 = "normal";
     SampleApp s2 = new SampleApp();
     boolean result =
                s2.isRegistered(s4, s5);
}
```

Figure 10: Test case after modification.

4.4 Phase 3: Execute Test Cases and Generate Test Result Summary

To detect SQLIAs, SQLUnitGen instruments application byte code with the AMNESIA runtime monitor [12]. When the SQL query constructed from a test case and the SQL query model built by AMNESIA statically do not match, the AMNESIA runtime monitor raises a SQLIAException.

To help programmers to easily identify vulnerable locations in the program, SQLUnitGen generates a textual test result summary and a graphical test result summary. A textual test result summary shows the test cases that succeeded or failed for each method. Test success in a test case means that SQLIA was not detected from the test case. Test failure in a test case means that SQLIA was detected from the test case.

A graphical test result summary shows a colored call graph indicating the flow of input be-

tween methods, and the demonstrated vulnerability of each method. On the call graph, methods are represented as ovals. A green oval indicates that all the test cases for the method succeeded. A red oval indicates that all the test cases for the method failed. A yellow oval indicates that some of the test cases for the method succeeded and that some of them failed. A black oval indicates there were no test cases for the method. From the colored call graph, developers can identify how vulnerable input flows through the system and where to put input filters. Figure 11 shows an example of colored call graph. For ease of reading a printed document, the color has been changed to black and white. The color mapping is described in the Figure 11.

In the figure, the numbers after a method name indicates the number of successful test cases and failed test cases. For example, all the test cases for `Login` succeeded, which means the method is not vulnerable to the test input. All the test cases for `isRegistered` failed, which means the method was vulnerable to the test input. Partial success can happen, as in `getUserInfo`, when input filtering is performed only for some of the arguments, or input filter does not check some of vulnerable characters.

Even though `isRegistered` failed in all the test cases, `isRegistered` can be considered not to be vulnerable because the caller method `Login` is the only path to `isRegistered`, and `Login` filtered all the vulnerable input successfully. If `isRegistered` can be called with user input without passing through `Login`, a programmer should consider putting input filters between `isRegistered` and the hotspot.

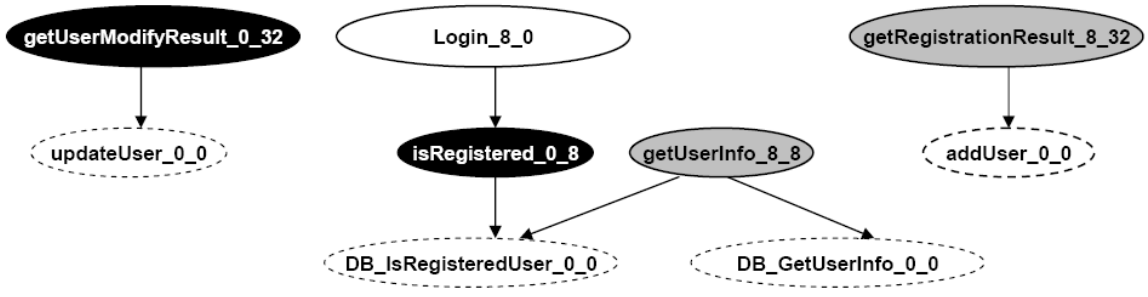
A textual test result summary also provides the query used for each test case. Therefore, programmers can easily identify why the test case failed and which user input is vulnerable using the textual test result summary and attack test cases with concrete attack input values. Based on the information, programmers can add input filters in a proper location in a program or use safer APIs, or combine the two approaches.

4.5 Attack Patterns

Our initial set of nine attack patterns included in SQLUnitGen v0.5 were collected from published black lists [2, 17, 19, 21]. These initial nine are only a small set of commonly-referenced attacks. To generate attack inputs, we used the following attack patterns for character type variables that require a single quotation mark:

```
AP1: 1’ OR ‘1’=’1
AP2: 1’OR‘1’=’1
AP3: 1'; exec master..xp_cmdshell
       'dir';--
AP4: 1'; drop table sqliatest; create
    table sqliatest (name varchar(10))--
AP5: 1'; delete from sqliatest;--
```

AP2 is similar to AP1, but AP2 does not have a space between `1’` and `OR`. This attack pattern was added to test if the program tries to detect SQL injection attack based on a wrong regular expression, for example, “`’ OR`”. In some cases, the application cannot just block a single quotation mark, because a single quotation mark in a name (e.g. O’Reilly) is legitimate data. Even though the regular expression “`’ OR`” can detect AP1, this regular expression will fail to detect AP2. In MySQL and PostgreSQL, ‘`-- `’ makes



White oval with solid line: all pass (green), *Grey oval*: some pass/some fail (yellow)

Black oval: all fail (red), *White oval with dashed line*: no test case (black)

Figure 11: Call graph with test result.

remaining content in a SQL query a comment so that the remaining content is ignored.

We used the following attack patterns for integer type variables:

```
AP6: 1 OR 1=1--
AP7: 1;exec master..xp_cmdshell  'dir'; --
AP8: 1; drop table sqliatest; create
     table sqliatest (name varchar(10)) --
AP9: 1; delete from sqliatest;--
```

User input from web interface is transmitted as string data types. Even though the input is for an integer type column in a database table, the string type user input can be used to construct a valid SQL query as long as the user input consists of numerical digits. Therefore, AP6-9 can be used as attack input value for an integer type column if an application does not convert the user input into the integer type.

AP1, AP2, AP6 are intended to detect attacks based on tautology as explained in Section 2.1. Some DBMSs allow multiple SQL statements in one query execution by default or by option. AP3, AP4, AP5, AP7, AP8, and AP9 are intended to detect attacks involving multiple SQL statements. AP4, AP5, AP8, and AP9 show that attackers can arbitrarily manipulate data in a database. With AP3, attackers can execute system commands using built-in stored procedure. Microsoft SQL Server is known to be vulnerable to AP3 [2]. For repeatable test, AP8 creates table `sqliatest` after the table is dropped.

4.6 Limitations

Although our approach is useful to test SQL injection vulnerabilities, the current implementation has limitations as follows.

1. Attack patterns in the test cases

The ability to detect vulnerabilities in our approach is limited by the quality of attack patterns. Therefore, false negatives can happen when the attack patterns included in SQLUnitGen are not sufficient, as is the case with the nine current attack patterns. Due to the ever-evolving attack patterns, it is difficult to include all the attack patterns in the current implementation. Therefore, an extendible interface to easily add new attack patterns must be provided.

2. False negatives

False negatives also can happen in the following cases. First, false negatives can be generated when a SQL query model does not precisely reflect the possible SQL queries in an application and considers a SQL query constructed from attack input as legal user input [12]. Second, false negatives can occur when the execution of initially-generated test cases do not reach hotspots through every possible path due to the lack of proper input. In this case, the necessary attack test cases are not generated. Therefore, the colored call graph also may not include all the possible paths to hotspots. Third, false negatives also can happen when a SQL query fails and the test case exits without executing the remaining SQL queries that can reveal SQLIA.

3. User input APIs

Our approach only can test user input passed as method arguments. If a user input API and a hotspot that uses the input reside in the same method, test case cannot be generated properly. One way to solve this problem is to give warning to programmers when the user input is used at a hotspot without being passed as a method argument. Such user input can be identified when an augmented SQL query model has no tagged information for a variable in a SQL query.

4. Underlying techniques

Our approach is limited by the ability of query-model building of AMNESIA and its underlying string analyzer, JSA. SQLUnitGen inherits the possibility of false negatives and false positives of AMNESIA due to the over-approximation of underlying string analysis [12]. In addition, the current implementation of JSA does not support the analysis of the string in a class field as it does for local string variables for its own purpose [8]. Therefore, when a method argument is a class and a member field of the class is used to construct a SQL query, our approach does not trace user input precisely. SQLUnitGen is also limited by the scalability of AMNESIA and JSA due to the possible large number of states and transitions in the generated automata [12].

5 Evaluation

To investigate the effectiveness of our approach, we performed case studies on two small web applications. This is a preliminary study to examine and understand the possibility of using this approach as an improvement to current vulnerability detection methods, and to see what we can learn about improving this approach. The following

subsections describe our evaluation setup and the results.

5.1 Evaluation Setup

As test subjects, we used two small web applications, called Cabinet and Bookstore with SQLUnitGen v0.5. Cabinet was developed as a class project by the first author and her team members in Fall 2004. Cabinet has approximately 2000 lines of code in 14 classes. Cabinet allows users to register, login, and order cabinets. Bookstore, approximately 20000 lines of code in 28 classes, is publicly available from an open source web site[9]. Bookstore was also used for the evaluation of AMNESIA [12]. For both of the applications, we used only the login modules as initial test.

Due to Limitations 3 and Limitation 4 described in Section 4.6, we modified a part of the subjects. Because SQLUnitGen requires user input to be passed to a hotspot through a method call, we added a wrapper method for a hotpot when the input API and the hotspot reside in the same method. Because underlying tool JSA does not analyze the fields as it does for local variables, we also modified the subjects so that the method arguments whose value is used for a hotspot are passed as string types rather than as a field in a class.

In addition, to evaluate the effectiveness of generated test cases, we performed controlled fault injection. For the fault injection, we modified the applications so that the applications have different levels of input filtering. For each subject, Version 1 has no input filtering function. Version 2 has input filtering for a part of input arguments. Version 3 was intended to have exhaustive input filtering for every input from users. The modified subjects are available online[10]. Cabinet initially had no input filtering in the server-side code. All input validation was implemented in the client side via Javascript.

5.2 Evaluation Results

We compared our results with the results of a static analysis tool, FindBugs [14]. FindBugs is a tool that detects various bug patterns in Java programs, including SQLIAs. FindBugs gives a warning when a SQL query is constructed from variables, not purely from constant values. We used MySQL11 v5.0.21 for the evaluation.

Table 1 shows the comparison summary at the hotspot level. Table 2, in the Appendix, provides a summary from a test case perspective and additional explanatory details. In Table 1, the numbers beside the subject name are the version numbers indicating different levels of input filters, as described in Section 5.1. A limitation of the evaluation is the small sample size of hotspots. In our future work we will evaluate larger programs.

For the comparison, we measured the number of false positives and false negatives. False positives are vulnerabilities found when the vulnerabilities do not exist. False negatives are vulnerabilities that were not found when the vulnerabilities actually exist. The false positives and false negatives are measured by counting the number of actual vulnerable hotspots in each application and the number of errors or warnings from SQLUnitGen and FindBugs. A vulnerable hotspot is a hotspot where a SQLIA can occur from the attack patterns defined in Section 4.5 via the execution through all possible paths starting from the top level of method call chain. Therefore, for SQLUnitGen, a vulnerable hotspot is a hotspot whose top level caller is not a green color in the colored call graph. However, due to the Limitation 2 in Section 4.6, we manually inspected the vulnerable hotspots. When a syntactically-correct SQL query at a hotspot cannot be created successfully from any method call, we did not count the hotspot vulnerable. For example, in Bookstore, one method includes only a part of a SQL query that is impossible to execute the query without a syntax error from any method call in the class we tested.

Bookstore Version 1 and Cabinet Version 1 had one and five vulnerable hotspots, respectively. Bookstore Version 3 and Cabinet Version 3 had no vulnerable statements because all of the input was properly filtered. FindBugs had no false negatives for all vulnerable hotspots. However, FindBugs had high percentage of false positives in Cabinet Version 2 and Cabinet Version 3. The reason FindBugs has false positives is because FindBugs only analyze the hotspots without considering the execution flow of the methods that call the hotpots and, therefore, cannot find the

[9] http://www.gotocode.com

[10] http://www4.ncsu.edu/~yshin2/sqlunitgen/examples.zip

[11] http://www.mysql.com

Applications	Tools	Vulnerable hotspots	Vulner-abilities found	False positives	False negatives
Bookstore 1	SQLUnitGen	1	1	0 (0%)	0
	FindBugs	1	1	0 (0%)	0
Bookstore 2	SQLUnitGen	1	1	0 (0%)	0
	FindBugs	1	1	0 (0%)	0
Bookstore 3	SQLUnitGen	0	0	0 (0%)	0
	FindBugs	0	1	1 (100%)	0
Cabinet 1	SQLUnitGen	5	3	0 (0%)	2
	FindBugs	5	5	0 (0%)	0
Cabinet 2	SQLUnitGen	1	1	0 (0%)	0
	FindBugs	1	5	4 (80%)	0
Cabinet 3	SQLUnitGen	0	0	0 (0%)	0
	FindBugs	0	5	5 (100%)	0

Table 1: Comparison with static analysis tool.

filter. SQLUnitGen had no false positives for all the vulnerable hotspots. However, SQLUnitGen had two false negatives in Cabinet 1. The false negative was caused because two vulnerable SQL statements were in the same method and the method returned after the first SQL statement was executed without executing the second statement. We will address this limitation in our future work.

6 Conclusions and Future Work

We presented an automatic test case generation technique to identify SQL injection vulnerability. We used a combined approach of static analysis, runtime detection and automatic testing. Static analysis using AMNESIA and string argument instrumentation is used to trace the user input to a vulnerable location (hotspot) in an application. In our evaluation, the prototype tool SQLUnitGen v0.5 had no false positives and small number of false negatives. Many of the SQLIA vulnerabilities that can be identified with SQLUnitGen can be addressed via the use of `PreparedStatements`. Since many legacy applications were not written using `PreparedStatements`, SQLUnitGen can be particularly helpful in finding and addressing SQLIA vulnerabilities in legacy code as well as new code written that does not take advantage of newer, safer techniques.

Due to the limitations that we described in Section 4.6, we could not perform large scale evaluation. However, the result of preliminary study shows that our technique is promising in detecting vulnerabilities with less false positives than static analysis tools. However, due to the possible false negatives of our tool, the two approaches can be used in a complementary way. That is, after detecting vulnerabilities with a static analysis tool such as FindBugs, we can check the real vulnerabilities with our tool.

Our future work includes the following: First, we plan to investigate a new test case generation technique with high path coverage. We can adapt symbolic execution approach combined with concrete execution [24] or infer the input values from existing manually-created test cases to reduce false negatives. Second, we plan to apply our approach for other types of security vulnerabilities. Our approach can be applied to the vulnerabilities with known hotspots where string type user input is used. Third, we plan to provide test coverage information to give high confidence on the testing results. Fourth, we plan to construct attack pattern database with more thorough attack input in a way new attack pattern can be easily added and used to generate test cases.

Acknowledgements

This work is supported in part by the National Science Foundation under CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Alessandro Orso and William Halfond provided us with an implementation of their

AMNESIA tool and heartfelt support to configure it in our system. Zhendong Su and Gary Wassermann, Koushik Sen, and Christoph Csallner, Anders Møller provided prompt and thorough answers to queries about their tools. We would like to thank the NCSU Software Engineering Realsearch group for their careful review of and helpful suggestions for the paper.

About the Author

Yonghee Shin is a Ph.D. student in North Carolina State University. Her research interest is in software reliability and software security testing. Her e-mail address is yonghee.shin@ncsu.edu.

Dr. Laurie Williams is an assistant professor in North Carolina State University. Her research area is software reliability, software security testing, pair programming, and extreme programming. Her e-mail address is williams@csc.ncsu.edu.

Dr. Tao Xie is an assistant professor in North Carolina State University. His research area is in automatic software testing. His e-mail address is xie@csc.ncsu.edu.

References

[1] "OWASPD – Open Web Application Security Project. Top ten most critical web application vulnerabilities," 2005.

[2] C. Anley, Advanced SQL Injection In SQL Server Applications. White Paper, Next Generation Security Software Ltd., 2000. http://www.ngssoftware.com/papers/advanced_sql_injection.pdf.

[3] B. Arkin, S. Stender, and G. McGraw, "Software Penetration Testing," *IEEE Security and Privacy*, vol. 3, no. 1, pp. 84 - 87, 2005.

[4] L. Auronen, "Tool-Based Approach to Assessing Web Application Security," Helsinki University of Technology, November, 2002.

[5] R. A. Baker, "Code Reviews Enhance Software Quality," *In Proceedings of the 19th international conference on Software engineering (ICSE'97)* pp. 570 - 571, Boston, MA, USA. 1997.

[6] P. Chandra, B. Chess, and J. Steven, "Putting the Tools to Work: How to Succeed with Source Code Analysis," *IEEE Security and Privacy*, vol. 4, no. 3, pp. 80-83, 2006.

[7] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security and Privacy*, vol. 2, no. 6, pp. 76 - 79, 2004.

[8] A. S. Christensen, A. Mller, and M. I. Schwartzbach, "Extending Java for high-level Web service construction," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 6, pp. 814--875, 2003.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise Analysis of String Expressions," *In Proceedings of the 10th International Static Analysis Symposium(SAS 03)*, pp. 1–18, June. 2003.

[10] W. R. Cook and S. Rai, "Safe Query Objects: Statically Typed Objects as Remotely Executable Queries," *In Proceedings of the 27th international conference on Software engineering (ICSE'05)*, St. Louis, Missouri, USA, May 15–21. 2005.

[11] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software -- Practice and Experience*, vol. 34, no. 11, pp. 1025-1050, 2004.

[12] W. G. J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," *In Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE)*, pp. 174 - 183, Long Beach, CA, USA, November. 2005.

[13] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code*: Addison Wesley, 2004.

[14] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 132-136, October. 2004.

[15] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA: Microsoft Press, 2003.

[16] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," *In Proceedings of the 12th international conference on World Wide Web*, pp. 148 - 159, Budapest, Hungary. 2003.

[17] S. Joshi, SQL Injection Attack and Defense, 2005. http://www.securitydocs.com/library/3587.

[18] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A Web Vulnerability Scanner," *The 15th World Wide Web Conference (WWW'06)*, Edinburgh, Scotland, May 23–26. 2006.

[19] S. Kost, "Introduction to SQL Injection Attacks for Oracle Developers," Integrity Corporation, January, 2004.

[20] O. Maor and A. Shulman, Blindfolded SQL Injection, Imperva inc., 2003. http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html.

[21] O. Maor and A. Shulman, SQL Injection Signatures Evasion: An overview of why SQL Injection signature protection is just not enough, 2004. http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html.

[22] R. A. McClure and I. H. Krüger, "SQL DOM: Compile Time Checking of Dynamic SQL Statements," *In Proceedings of the 27th international*

conference on Software engineering (ICSE'05), pp. 88 - 96, St. Louis, Missouri, USA., May 15–21. 2005.

[23] G. McGraw, *Software Security: Building Security In*: Addison-Wesley Software Security Series, 2006.

[24] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pp. 263-272, Lisbon, Portugal, September. 2005.

[25] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'06)*, pp. 372 - 382, Charleston, South Carolina, USA, January 11-13. 2006.

[26] F. Swiderski and W. Synder, *Threat Modeling*: Microsoft, 2004.

[27] F. Valeur, D. Mutz, and G. Vigna, "A Learning-Based Approach to the Detection of SQL Attacks," *In Proceedings of the Conference on Detection of Intrusions and Malware Vulnerability Assessment(DIMVA)*, July. 2005.

[28] W. D. Yu, P. Supthaweesuk, and S. Aravind, "Trustworthy Web Services Based on Testing," *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pp. 167 - 177, Shanghai, China. 2005.

Appendix

In this appendix, we present additional evaluation results to the evaluation results described in Section 5.2. Table 2 describes the number of hotspots (H), the number of initially generated test cases (IT), hotspot-reaching test cases (HT), and generated attack test cases (AT) for each application. In addition, Table 2 includes the number of successful SQLIAs and SQL exceptions.

Because different DBMSs and database drivers behave differently for the SQL query or for the same database interface APIs in some cases, we tested with two DBMSs, MySQL[12] v5.0.21 and PostgreSQL[13] v8.1.3. For example, MySQL does not allow multiple SQL statements in an SQL execute command by default, but PostgreSQL allows multiple SQL statements for certain APIs. To test the effects of multiple SQL statements, we modified SQL APIs used in the applications so that the APIs allow multiple SQL statements. For that purpose, `executeQuery` was changed to `execute`.

In Table 2, SI indicates the number of test cases with successful SQL injections without SQL syntax or semantic errors. FSI, *full path SQLIA*, indicates the number of test cases with successful SQL injections from the top level method in the method call chain. SI counted the number of successful SQLIA in terms of each method not considering the flow of input. FSI considered the flow of input. For example in Figure 11, even though `isRegistered` failed in all the test cases, `isRegistered` can be considered not to be vulnerable because the caller method `Login` is the only path to `isRegistered`, and `Login` filters all the vulnerable input successfully.

SQ indicates the number of test cases with SQL syntax and semantic errors. SQL syntax errors occur when attack input includes syntactically illegal input. SQL syntax error depends on the DBMS used in some cases. For example, multiple SQL statements cause a syntax error for MySQL, but not for PostgreSQL. For PostgreSQL, using a non-boolean expression in `WHERE` clause is a syntax error. However, MySQL allows non-boolean expressions in `WHERE` clause such as "`WHERE 1`". A SQL semantic error occurs when a generated SQL query uses incorrect table name or column name.

In Bookstore, PostgresSQL has more full path SQLIAs than MySQL because PostgreSQL allows multiple SQL statements and the attack patterns we used include multiple SQL statements. In Cabinet, MySQL has more full path SQLIAs than PostgreSQL because PostgreSQL treated some SQL statements as illegal statements, when MySQL considered them legal, as described previously. Due to the different level of input filtering, Bookstore 1 and Cabinet 1 had more successful SQLIAs than Bookstore 2 and Cabinet 2, respectively. In Bookstore 3 and Cabinet 3, all of the SQLIA prevented in terms of FSI due to the input filtering.

	Test case information				MySQL			PostgreSQL		
Subject	**H**	**IT**	**HT**	**AT**	**SI**	**SQ**	**FSI**	**SI**	**SQ**	**FSI**
Bookstore 1	1	9	52	1	18	77	9	40	55	13
Bookstore 2	1	9	52	1	14	76	6	36	54	9
Bookstore 3	1	10	52	1	12	73	0	30	55	0
Cabinet 1	5	348	142	5	24	24	21	23	25	20
Cabinet 2	5	348	142	5	15	15	12	14	16	11
Cabinet 3	5	348	142	5	0	0	0	0	0	0

Table 2: Test result.

H: Number of hotspots. **IT**: Number of initially generated test cases in the phase 1. **HT**: Number of hotspot-reaching test cases. **AT**: Number of attack test cases. **SI**: Number of successful attacks. **SQ**: Number of SQL syntax or semantic errors. **FSI:** Number of full path SQLIA.

[12] http://www.mysql.com

[13] http://www.postgresql.org