

# A Static Analysis Framework For Detecting SQL Injection Vulnerabilities

Xiang Fu   Xin Lu   Boris Peltsverger   Shijun Chen  
School of Computer and Information Sciences  
Georgia Southwestern State University, Americus, GA 31709

Kai Qian  
School of Computing and Software Engineering  
Southern Polytechnic State University, Marietta, GA 30060

Lixin Tao  
Computer Science Department  
Pace University, Pleasantville, NY 10570

## Abstract

Recently *SQL Injection Attack (SIA)* has become a major threat to Web applications. Via carefully crafted user input, attackers can expose or manipulate the back-end database of a Web application. This paper proposes the construction and outlines the design of a static analysis framework (called *SAFELI*) for identifying SIA vulnerabilities at compile time. *SAFELI* statically inspects MSIL bytecode of an ASP.NET Web application, using symbolic execution. At each hotspot that submits SQL query, a hybrid constraint solver is used to find out the corresponding user input that could lead to breach of information security. Once completed, *SAFELI* has the future potential to discover more delicate SQL injection attacks than black-box Web security inspection tools.

**keywords:** SQL Injection Attack, Symbolic Execution, Constraint Solver, Automatic Testing.

## 1 Introduction

Web applications have been very successful in E-Commerce for almost two decades. However, many Web applications today suffer from SQL Injection Attack (SIA) [1, 2], when they construct SQL queries on the fly based on user input. Attackers may be able to trick the back-end database of a Web application into executing malicious SQL code. This allows to expose, manipulate, and destroy the back-end database, hence causing great losses. Since SIA is conducted at application level, normal firewall and intrusion detection at network layer have no defense against SIA. Consider the following well-known example [1].

**Example 1.1** A log-in page has two text box fields for entering user name and password. Let `sUname` and `sPwd` represent the strings contained in text boxes. Presented below is a piece of back-end C# code that constructs SQL statement on the fly. Here “+” denotes string concatenation.

```
"SELECT uname, pass FROM users WHERE \n uname=" +  
+ sUname + "' AND pass=" + sPwd + "'"
```

The above SQL query intends to verify existence of the user name/password pair supplied. However, if attacker enters “admin’ -- ” as user name, and leaves password blank, the following SQL statement is constructed.

```
SELECT uname,pass FROM users WHERE  
uname='admin' -- ' AND pass=''
```

Since “--” comments out the rest of query, it only verifies existence of user name “admin”. The attacker can log in without supplying password as long as admin account exists. ■

One typical approach against SIA is to filter out special characters such as single quote and “--”. However, it does not work for more delicate variations, e.g., the attack against integer columns in database [1]. Recently, many solutions are proposed to capture and defend SIA.

- *Tainted Data Tracking:* The main idea [15] is to track the data that comes from user input. This can be done via instrumenting the run time environment or interpreter of the back-end scripting language. When an SQL statement is submitted, its syntax tree is first examined. if any of its SQL keywords is identified to be from user input, the SQL statement is stopped.

- *Intrusion Detection Based on Static Analysis:* Using static string analysis technique [6], it is possible to construct a regular expression that conservatively approximates the set of SQL statements generated at a hotspot (which submits SQL query). The information can be used to statically analyze syntax correctness of SQL statements [8] and to model “normal behaviors” of a Web application. During run time, any SQL statement not contained in the approximation library will be rejected by intrusion detection [9].
- *Black-box Testing:* Black-box testing [10, 7] can be used to discover SIA vulnerabilities, by applying a library of pre-collected attack patterns. It is fast and effective, however, without prior knowledge of source code, it has difficulty in discovering non-trivial vulnerabilities.
- *SQL Randomization:* As an extension of instruction randomization [11], SQL randomization [3] instruments a Web application and appends a random number after each SQL keyword used to build SQL statements. SQL parser is modified correspondingly to accept randomized SQL keywords. At run-time, since a user injected SQL keyword does not have random number appended, SIA fails due to syntax error.

This paper proposes the construction of a static analysis framework (called SAFELI) for discovering SIA vulnerabilities at compile-time. While the tool is still under development, it is beneficial to share its main design idea in this paper. The contributions of this paper are listed below:

1. *White-box Static Analysis:* SAFELI analyzes bytecode. It relies on string analysis (similar to [9]). However, it discovers vulnerabilities at compile-time, but not at run-time. Once implemented, SAFELI is able to generate user inputs as hard-evidence of a vulnerability.
2. *Hybrid-Constraint Solver:* SAFELI employs a brand new string analysis technique (other than the one used in [6, 9, 8]). The technique can handle hybrid constraints that involve boolean, integer, and string variables. Most popular string operations can be handled.

This paper is organized as follows. Section 2 presents a motivating example to be used throughout the paper. Section 3 introduces the general structure of SAFELI. Section 4 discusses the symbolic execution framework. Section 5 presents the core technique, i.e., hybrid constraint solver. Section 6 covers test case generation and applies hybrid constraint solver algorithm on the motivating example. Section 7 concludes the paper and proposes the future work.

## 2 Motivating Example

This section describes a non-trivial SIA vulnerability that can not be easily detected by black-box testing tools. It is motivated by the vulnerability example of string size restriction in [1]. Fig. 1 presents a `message()` method that processes a user input string in two steps. First, it checks whether the input string contains suspicious SQL keywords such as “--”, “OR”, and “drop”. Then it substitutes each single quote character with “'”, i.e., escape character of single quote in SQL. Finally, `message()` tries to provide further protection by restricting the length of the output within 16.

```
String message(String strInput)
{
    //1. SQL keyword search
    if(strInput.IndexOf("--")!=-1
        || strInput.IndexOf("OR")!=-1
        || strInput.IndexOf("drop")!=-1)
        throw new Exception(
            "Possible SQL Injection Attack: " + strInput
        );

    //2. message the data for single quote
    String sOut = strInput.Replace("'", "'");
    sOut = sOut.Substring(0,16);
    return sOut;
}
```

Figure 1. String Message

The SQL generation statement in Example 1.1 can be strengthened as below.

```
"SELECT uname, pass FROM users WHERE \n uname=" +
+message(sUname)+ "' AND pass=" +message(sPwd)+ "'"
```

Readers can verify that the above code can defend the attack strings in Example 1.1 because “--” is filtered out. Further more, single quote characters will be replaced by “'” and hence causing no harm. The `message()` function in Fig. 1, however, has a very delicate bug. Consider the following input for user name and password, respectively:

```
123456789012345'
OR      uname<>'
```

Notice both strings are 16 characters long. After going through the message operations, the following SQL statement is generated.

```
SELECT uname,pass FROM users WHERE
uname='123456789012345' AND pass=' OR      uname<>'
```

Notice that its WHERE clause is always a tautology. It consists of two conditions: (1) whether `uname` is equal to constant string “123456789012345' AND pass=” (note the escape character “'” inside), or (2) `uname` is not an empty string. Obviously “`uname<>'`” always evaluates to true. The trick is that the length of malicious strings

are both 16. Although the `Replace` function generates escape characters “'”, half of the “'” is then cut off by the `Substring` method at the end of `message()`. Such delicate bugs cannot be easily discovered by black-box testing tools.

### 3 SAFELI Framework

This section presents the overall structure of SAFELI (Static Analysis Framework for discovering sql Injection vulnerabilities), which is currently under development. SAFELI consists of the following components:

- *MSIL Instrumentor*: The module instruments MSIL bytecode of an ASP.NET application for symbolic execution. It inserts additional monitoring function at each location where data members of objects are accessed. Values of (uninitialized) variables are replaced with symbolic constraints. Each hotspot, i.e., the location which submits SQL statement, is tagged to trigger constraint solver.
- *Symbolic Execution Engine*: It is essentially a wrapper of the .Net Framework. The execution engine iteratively examines the back-end code for each ASP page one by one. When hotspots are reached, a library of pre-set attack patterns is consulted, based on which a hybrid string constraint is constructed and sent to constraint solver for generating vulnerability evidence (user input).
- *Library of Attack Patterns*: The module stores a collection of pre-set attack patterns, each of which is represented using a regular expression.
- *Constraint Solver*: Given a constraint, the solver tests its satisfiability and generates valuation of variables that satisfy the constraint. Different than other popular platforms of symbolic execution, the Constraint Solver of SAFELI can solve string constraints. Details are discussed in Section 5.
- *Test Case Generator*: When initial valuations are generated, they are passed to the Test Case Generator. The module then injects the values into HTML fields and posts the web page back to server. It then uses a heuristic algorithm to analyze the response from server. When vulnerability is verified, a step by step error trace is generated.

### 4 Symbolic Execution

This section presents the Symbolic Execution Engine of SAFELI. After a brief background introduction, we discuss the MSIL instrumentor of SAFELI.

#### 4.1 Background Information

The history of symbolic execution, which symbolically interprets and verifies correctness of sequential programs, can be dated back to 1970’s [14]. During symbolic execution, initial values of input variables are represented using symbolic constraints. Each branch in the program is tagged with a corresponding path condition. When an exception is encountered or some system safety property is violated, the path condition is sent to a constraint solver for generating the corresponding initial values of input variables. Symbolic execution has been widely applied in automatic test case generation [17], discovery of Operating System vulnerabilities [18], and combination of model checking to analyze heap configurations and data structures [13].

We briefly introduce the idea of symbolic execution using one example presented in Fig. 2. Function `PointInRectangle()` checks if a point  $(x, y)$  is contained in a rectangle whose left top vertex is  $(x_1, y_1)$  and whose width and height are  $w_1$  and  $h_1$  respectively. Function `hasCollision()` tests whether two rectangles collide with each other by examining whether any of the four vertices of the first rectangle is contained in the second rectangle. We verify the correctness of `hasCollision()` in the `main()` function by calling it twice and swapping the sequence of two rectangles in its input parameters. We expect `main()` function returns without any exception.

Symbolic execution starts at line 19, where all eight integer variables are assigned a symbolic value, and let them be  $a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2$ . Then symbolic execution traces to line 21 and steps into `hasCollision()` where

```

1: static bool PointInRectangle(int x, int y,
2:     int x1, int y1, int w1, int h1)
3: {
4:     return (x>=x1 && y>=y1 && x<=x1+w1 && y<=y1+h1);
5: }
6:
7: static bool hasCollision(int x1,int y1,int w1,int h1,
8:     int x2, int y2, int w2, int h2)
9: {
10:  bool b1 = PointInRectangle(x1,y1,x2,y2,w2,h2);
11:  bool b2 = PointInRectangle(x1+w1,y1,x2,y2,w2,h2);
12:  bool b3 = PointInRectangle(x1,y1+h1,x2,y2,w2,h2);
13:  bool b4 = PointInRectangle(x1+w1,y1+h1,x2,y2,w2,h2);
14:  return (b1 || b2 || b3 || b4);
15: }
16:
17: static void main(string [] args)
18: {
19:  int x1,y1,w1,h1; //uninitialized, or init by
20:  int x2,y2,w2,h2; //e.g., x1=int.Parse(args[0])
21:  bool b11 = hasCollision(x1,y1,w1,h1,x2,y2,w2,h2);
22:  bool b21 = hasCollision(x2,y2,w2,h2,x1,y1,w1,h1);
23:  if(b11==b21)
24:      return;
25:  else
26:      throw new Exception("hasCollision() incorrect!");
27: }

```

Figure 2. Collision Detection

four variables  $b_1$ ,  $b_2$ ,  $b_3$ , and  $b_4$  are added into variable collection. At line 10,  $b_1$  is associated with the following symbolic constraint, letting it be  $B_1$ :

$$a_1 \geq a_2 \wedge b_1 \geq b_2 \wedge a_1 \leq a_2 + c_2 \wedge b_1 \leq b_2 + d_2.$$

Similarly,  $b_2$ ,  $b_3$ , and  $b_4$  can be associated with symbolic constraints and let them be  $B_2$ ,  $B_3$ , and  $B_4$ . When symbolic execution returns to line 21,  $b_{11}$  is associated with  $B_1 \vee B_2 \vee B_3 \vee B_4$ .  $b_{12}$  is similarly assigned, and let it be  $B'_1 \vee B'_2 \vee B'_3 \vee B'_4$ . When the branch statement (line 23) is encountered, both branches are associated with a path condition. If the second branch is taken, the exception triggers the string solver to solve path condition  $B_1 \vee B_2 \vee B_3 \vee B_4 \neq B'_1 \vee B'_2 \vee B'_3 \vee B'_4$ . An integer solver like Omega Library [12] can easily decide that the above constraint is satisfiable. For example, one concrete valuation for  $(x_1, y_1, w_1, h_1, x_2, y_2, w_2, h_2)$  is  $(1, 1, 1, 1, 0, 0, 3, 3)$ .

## 4.2 Instrumentation

The instrumentor of SAFELI is still under development. This subsection describes its major design idea. Before going through Symbolic Execution Engine, MSIL code of an ASP.NET Application has to be instrumented. SAFELI relies on RAIL [5] for manipulating MSIL code, because RAIL provides the capability of replacing types, references, variables, and methods. Based on RAIL, the job of bytecode instrumentor is to inspect MSIL code, locate the get and set operations on each attribute/property and variable, and replace each variable access with a corresponding operation that constructs symbolic constraints. Before each hotspot that issues an SQL statement, e.g., the

```
class Constraint{
    public RelationalOp relOp;
    public virtual void resolve();
    public Object [] childNodes;
}
class Expr{...}
class BoolExpr: Expr{...}
class IntExpr: Expr{
    public IntExpr addWith(IntExpr expr);
    public IntExpr multiplyWith(IntExpr expr);
    ...
}
class StrExpr: Expr{
    public StrExpr Substring(int startIdx, int length);
    public StrExpr Replace(String sOld, String sNew);
    ...
}
class SymbolicState{
    ArrayList lstVars;
    int Loc;
    HashTable Map; //mapping from var to expr
    Constraint PathCond;
}
```

Figure 3. Classes Used In Instrumentation

```
String sqlstat = "";
if(a<2){
    sqlstat = "SELECT * FROM users WHERE uname='"+
        + txtboxUname.Text + "'";
}else{
    sqlstat = "SELECT * FROM users WHERE uname='guest'";
}
```

Figure 4. Sample Before Instrumentation

call of “`SqlCommand.ExecuteDataset()`”, an additional “`attack_test()`” function is called to iteratively test each malicious attacking pattern and find out initial values of input variables.

As shown in Fig. 3, in SAFELI a symbolic constraint is represented by a class called `Constraint`. A constraint is composed of basic elements ranging from integer expressions, string expressions, and hybrid expressions such as `sqlStatement.Length<5`. Thus the `Expr` class has two derived classes: `IntExpr` (i.e., integer expression) and `StrExpr` (i.e., string expression). `IntExpr` supports integer operations such as addition, subtraction, multiplication with constants, etc. Frequently seen string operations such as `Substring`, `Replace`, `IndexOf`, and `CharAt` are supported by `StrExpr`. To resolve pure integer constraints, SAFELI relies on the Omega library [12]. To resolve pure and hybrid string constraints, we use a unique string solver, which is presented in Section 5. The `SymbolicState` class embodies information of a symbolic state: a list of variables, current location, a hash table which maps from variables to symbolic expression, and a path condition associated with the current execution path.

We illustrate the idea and the expected effects of SAFELI bytecode instrumentor (currently under development) by an example in Fig. 4. The code snippet in Fig. 4 dynamically generates an SQL statement based on the value of an integer variable “`a`”. After instrumentation, the resulting instrumented code is displayed in Fig. 5.

```
1 static SymbolicState ss;
2 ...
3 ss.Map[sqlstat] = new StrExpr("");
4 if(ss.random_choice()){
5     ss.PathCond.AndWith(new Constraint(RelOp.LessThan,
6         new Object[] {ss.Map[a], new IntExpr(2)}));
7     ss.Map[sqlstat] = (new StrExpr("SELECT ... uname="))
8         .append(ss.Map[txtboxUname.txt])
9         .append("");
10 }else{
11     ss.PathCond.AndWith(
12         (new Constraint(RelOp.LessThan,
13             new Object[] {ss.Map[a], new IntExpr(2)})
14             ).Inverse()
15     );
16 ss.Map[sqlstat] = new StrExpr("SELECT ... guest'");
17 }
```

Figure 5. Sample After Instrumentation

In Fig. 5, at the “global” level, there is one instance of `SymbolicState`, named `ss`. Here “global” refers to the scope of the C# class that corresponds to the ASP page it serves. Each assignment of a variable is replaced by the statement that constructs an expression and updates the mapping of the symbolic state. For example, the “`sqlstat=""`” statement is translated into “`ss.Map[sqlstat]=new StrExpr(“”)i;`”.

The condition in each if statement (and similarly while loop) is replaced by a `random_choice()` function which non-deterministically generates a boolean value. At the beginning of each branch, the current path condition is updated correspondingly. For example, at line 5 of Fig. 5, i.e., beginning of the “true” branch, the path condition is conjuncted with the constraint that corresponds to the if-condition “`a<2`”.

The design of SAFELI bytecode instrumentor and symbolic execution engine generally follows the idea of S. Khurshid *et al.*’s work [13] on Java bytecode. Notice that in SAFELI we need to execute code multiple times to achieve the complete branch coverage, e.g., to execute the code in Fig. 5 twice guarantees a 50% coverage rate, and to execute it three times guarantees 75%. This naive instrumentation algorithm can be further improved.

## 5 Constraint Solver

This section outlines the algorithm of constraint solver module (currently under development). It has two responsibilities: (1) to decide satisfiability of path constraints, and (2) to find out the initial values of input variables that lead to the breach of database security. Notice that our algorithm is conservative w.r.t. satisfiability – it can report false negatives (i.e., a satisfiable constraint is reported as non-satisfiable) but will not report false positives. Its implications are as follows: (1) If SAFELI reports an error, the concretized input variable values will eventually lead to the error if the program is executed as many times as possible; and (2) If SAFELI does not report an error, there might still be vulnerability in the program because the constraint solver might have false negative reports.

### 5.1 Hybrid Constraint Solver

The design idea of hybrid constraint solver is very similar to that of the Action Language Verifier [19]. There are three categories of expressions in a hybrid constraint: boolean expression, integer expression, and string expression. Each type is represented symbolically. Boolean expressions are represented using BDD [16, 4], integer expressions are represented using Presburger constraints [12], and string expressions are represented using regular expression. A hybrid constraint is always represented in Disjunc-

tive Normal Form (DNF), as shown below:

$$\bigvee_i I_i \wedge B_i \wedge S_i.$$

In the above formula,  $I_i$ ,  $B_i$  and  $S_i$  represent the integer, boolean, and string expression in the  $i$ ’th conjunction. Notice that the set of variables appeared in  $I_i$  and  $B_i$  are mutually exclusive, however, integer variables in  $I_i$  could appear in  $S_i$ . For example, consider the following constraint (let it be  $S_1$ ):

$$i < 5 \wedge j > 2 \wedge \text{str1} = \text{str2.Substring}(i, j)$$

In this case, we have to solve integer and boolean constraints first, and then concretize the solution and substitute the appearance of any variables in  $S_i$  with the corresponding concretized values. We can generate multiple sets of solutions, and spawn multiple instances of the string constraint. For example, the following are part of the constraints generated for  $S_1$  above. However, note that this approach can lead to false negative.

$$(i = 1 \wedge j = 3) \wedge \text{str1} = \text{str2.Substring}(1, 3) \\ \vee (i = 2 \wedge j = 4) \wedge \text{str1} = \text{str2.Substring}(2, 4)$$

### 5.2 Backward String Image Computation

Backward image computation is the key to solving string constraints. It is an essential concept in symbolic model checking. In the context of string manipulation, we define *backward image* as follows: given a set of strings  $R$  and a string operation  $f$  (e.g., `Substring` and `CharAt`), the backward image of  $R$  w.r.t.  $f$  is the maximal set of strings  $X$  where for each string  $s \in X : f(s) \in R$ . In SAFELI, both  $R$  and its backward image are expressed using regular expression. We now briefly describe the image computation algorithm for popular string operations.

- *string length*: The function returns the length of a string. Given the following equation

$$s.Length = k,$$

where  $k$  is an integer constant, the solution of  $s$  is  $\Sigma^k$ , where  $\Sigma$  is the alphabet. Similarly, given an inequality  $s.Length < k$  the solution of  $s$  is  $\Sigma^0 \mid \Sigma^1 \mid \dots \mid \Sigma^{k-1}$ . Given an inequality  $s.Length \geq k$  the solution of  $s$  is  $\Sigma^k \Sigma^*$ .

- *substring*: The `Substring( $a, b$ )` operation chops one substring from source string, starting at index  $a$ , with length  $b$ . Here  $a$  and  $b$  are two integer constants. Given a regular expression  $r$  and the equation as below

$$s.Substring(a, b) = r,$$

the solution of  $s$  is  $\Sigma^a(r \cap \Sigma^b)\Sigma^*$ . Here  $\cap$  represents the intersection of regular languages. It is obvious that the solution of  $s$  is also regular because regular language is closed under concatenation, intersection, union, complementation, difference, and substitution.

- *charat*: The  $\text{CharAt}(a)$  function returns the character at index  $a$ . Given the equation

$$s.\text{CharAt}(i) = r,$$

the solution of  $s$  is  $\Sigma^i(r \cap \Sigma^1)\Sigma^*$ .

- *replace (character)*: The  $\text{Replace}(a, b)$  function replaces every occurrence of  $a$  with  $b$  in a string, where  $a$  and  $b$  are two characters. Given the equation

$$s.\text{Replace}(a, b) = r,$$

the solution of  $s$  is  $s = r_{b/(a|b)}$  where  $b/(a|b)$  means to replace every appearance of  $b$  with  $(a|b)$ . For example, if  $s.\text{Replace}('a', 'b') = b^*c^+$  then  $s = (a|b)^*c^+$ .

- *replace (string)*: Given  $s.\text{Replace}(s_1, s_2) = r$ , where  $s_1$  and  $s_2$  are two constant strings. The solution of  $s$  is computed as follows: construct a finite state machine  $\mathcal{A}$  that accepts  $r$ . Determinize  $\mathcal{A}$  and let it be  $\mathcal{A}'$ . For each state pair  $c_1, c_2$  in  $\mathcal{A}'$  such that there is a path which could produce  $s_2$ , add another path (states and transitions) from  $c_1$  to  $c_2$  such that the string along the newly added path is  $s_1$ . The modified automata accepts the solution of  $s$ .

- *string concatenation*: Given  $r = s_1 + s_2$ , where  $s_1$  and  $s_2$  are unknown, solution of  $s_1$  is generated as follows: convert  $r$  to a finite state machine, now mark every state as a final state. The new automaton accepts the prefix language of  $r$ , i.e.,  $s_1$ .

$s_2$  can be solved as follows: given the automaton  $\mathcal{A}$  that accepts  $r$ , construct another automaton  $\mathcal{A}'$  from  $\mathcal{A}$  such that  $\mathcal{A}'$  accepts the reverse of  $r$  (i.e., every word accepted by  $\mathcal{A}'$  has its reverse in  $r$ ). This can be simply achieved by making each final state in  $\mathcal{A}$  an initial state in  $\mathcal{A}'$  and the initial state in  $\mathcal{A}$  the final state in  $\mathcal{A}'$  and then reversing the direction of all transitions. Similarly, construct the prefix automaton from  $\mathcal{A}'$ , and let it be  $\mathcal{A}''$ . Construct the reverse of  $\mathcal{A}''$  and that is the finite state machine accepting  $s_2$ .

Note that the above are “maximal” solutions of  $s_1$  and  $s_2$ . In practice, we often have to solve an equation like  $r = s + c$  where  $r$  is a regular expression and  $c$  is a constant string. Solving such equations is very useful in generating attack strings. Consider Example 5.1, which is a simplified version of the motivating example in Section 2.

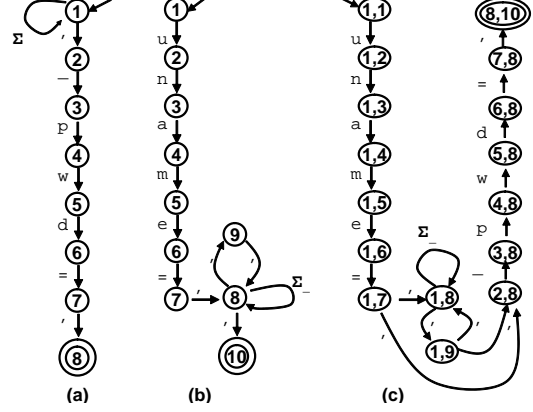


Figure 6. Solving String Equation

**Example 5.1** Let  $\Sigma$  represent the ASCII alphabet.  $\Sigma_-$  is defined as  $\Sigma_- = \Sigma - \{ '\}$  and  $\Sigma_+$  is defined as  $\Sigma_+ = \Sigma_- \cup \{ '\}$ . In another word,  $\Sigma_-$  excludes single quote, which is replaced by its SQL escape character in  $\Sigma_+$ . Consider the following equation, where  $\equiv$  is used to separate the left and right hands of the equation.

$$\text{"uname='"} + s + \text{"' p w d = '"} \equiv \text{"uname='"} \Sigma_+^* \text{'}$$

Its intuition is essentially to ask: if we concatenate the two constant strings (which are intended to test two data columns `uname` and `pwd`) with  $s$ , is it possible to generate one single condition that tests on column `uname` only? To solve the above equation, we can first solve the following:

$$s' + \text{"' p w d = '"} \equiv \text{"uname='"} \Sigma_+^* \text{'}$$
 (1)

Then we can get  $s$  from:

$$\text{"uname='"} + s \equiv s'$$
 (2)

To solve Equation (1), we need to compute the intersection of two strings:  $\text{"* * ' p w d = '"} (let it be  $s_a$ ) and  $\text{"uname='"} \Sigma_+^* \text{'}$  (let it be  $s_b$ ). The finite state automata accepting  $s_a$  and  $s_b$  are presented in Fig. 6 (a) and (b). Note that in Fig. 6, “-” indicates space character. The automaton accepting  $s_a \wedge s_b$  is displayed in Fig. 6(c). Then we study each state in Fig. 6(c), starting from which there is path of  $\text{"' p w d = '"} leading to the final state. Readers can verify that (1, 9) and (1, 7) are the only states that satisfy the condition. Then we mark (1, 9) and (1, 7) as final states, and unmark the original final state in Fig. 6(c). The resulting automaton accepts the solution of  $s'$ , which is expressed using a regular expression$$

$$\text{uname= | uname=' ( ' | \Sigma_- )^* \text{'}$$

Similarly we can solve Equation (2), and the regular expression solution of  $s$  is displayed below:

$$\text{( ' | \Sigma_- )^* \text{'}$$

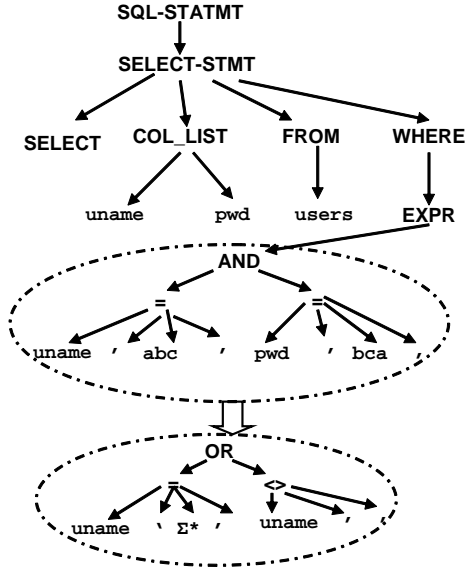


Figure 7. Solving Motivating Example

## 6. Test Case Generation

We now describe the idea of attack pattern library and test case generator which are currently under development. When symbolic execution reaches each hotspot, the test case generator randomly generates some commonly used strings (not malicious) and instantiates the dynamically constructed SQL statement. An abstract syntax tree is then constructed, where all table names, column names are known to the test case generator (e.g., `users`, `uname`, and `pwd` in Fig. 7). Then depending on the syntax tree, test case generator pulls from the attack pattern library a set of applicable attack patterns and parameterize them with the column names. The idea of most attack patterns is to enforce the `WHERE` clause to be tautology. For example, in Fig. 7, the expression used in the original `WHERE` clause is replaced by an attack pattern “`COLUMN = 'Σ+*' OR COLUMN<>'`”, where “`COLUMN`” is replaced by “`uname`” for the motivating example. Then the concrete test case is generated, as shown in the following example.

**Example 6.1** A set of equations can be established for the motivating example in Section 2, whose SQL statement structure is presented in Fig. 7. Note that “`Substr`” stands for “`Substring`” in the following equations.

$$s_1 = s_{\text{uname}}.\text{Replace}(" ", "'').\text{Substr}(0,16) \quad (3)$$

$$s_2 = s_{\text{pwd}}.\text{replace}(" ", "'').\text{Substr}(0,16) \quad (4)$$

$$\begin{aligned} & \text{"uname='"} + s_1 + \text{"' pwd='"} + s_2 + \text{"'"} \\ \equiv & \text{uname='}\Sigma_+^*'\_*\text{OR uname}<>' \end{aligned} \quad (5)$$

In the above equations,  $s_1$  and  $s_2$  are the massaged strings of the user name and password. Note that “`_*`” on the right of Equation 5 represents a sequence of white spaces. Using a similar technique in Example 5.1, we can get the solution of  $s_1$  as follows:

$$(''|\Sigma_+)^*$$

Replace  $s_1$  with its solution in Equation 5, we solve  $s_2$  and its solution is presented as below where “`_`” stands for white space.

$$\_*\text{OR uname}<>'$$

With  $s_1$  and  $s_2$ , we proceed to solve Equations 3 and 4. Using the algorithm to solve substitution and substring, we can easily get the solution of  $s_{\text{uname}}$  as follows<sup>1</sup>:

$$(''|\Sigma_+)^{15}\Sigma_+^*$$

The solution of  $s_{\text{pwd}}$  is expressed using a regular expression as below:

$$\text{OR uname}<>' \Sigma_+^*$$

where there are 5 spaces before “`OR`”.

Concretize the variables  $s_{\text{uname}}$  and  $s_{\text{pwd}}$  we can generate the evidence of vulnerability, as given in Section 2.

## 7 Conclusion

This paper has proposed and outlined the main design idea of SAFELI, a static analysis tool which can automatically generate test cases exploiting SQL injection vulnerabilities in ASP.NET Web applications. The novelty of the tool lies in its satisfiability decision/approximation procedure for string constraints. By symbolically executing an ASP.NET Web application, SAFELI constructs equations on strings which match a certain attack pattern. Once fully implemented, SAFELI can take advantage of source code information and will be able to discover very delicate vulnerabilities that cannot be discovered by black-box vulnerability scanners. Our future work includes completing the implementation of SAFELI and exploring algorithms to automatically enumerate SQL `WHERE` clauses.

## References

- [1] C. Anley. Advanced SQL Injection In SQL Server Applications. Next Generation Security Software LTD. White Paper, 2002.

<sup>1</sup>Note that the “15” in the formula actually does not mean to repeat  $(''|\Sigma_+)^{15}$  times, but to restrict the length of its repetitions to 15.

- [2] C. Anley. More Advanced SQL Injection. Next Generation Security Software LTD. White Paper, 2002.
- [3] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, volume 3089 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2004.
- [4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
- [5] B. Cabral, P. Marques, and L. Silva. RAIL: Code Instrumentation for .NET. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC)*, 2005.
- [6] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Static Analysis Symposium (SAS'03)*, 2003.
- [7] SPI Dynamics. Webinspect: Security throughout the application lifecycle. SPI Dynamics. Datasheet. [http://www.spidynamics.com/assets/documents/WebInspect\\_DataSheets.pdf](http://www.spidynamics.com/assets/documents/WebInspect_DataSheets.pdf).
- [8] C. Gould, Z. Su, and P. Devanbu. JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 697–698, 2004.
- [9] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software enginee*, pages 174–183, 2005.
- [10] 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 11th International World Wide Web Conference (WWW 2003)*, 2003.
- [11] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction set randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'03)*, 2003.
- [12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.
- [13] S. Khurshid, C. S. Pasăreănu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, 2003.
- [14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [15] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.
- [16] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1986.
- [17] N. Tillmann and W. Schulte. Parameterized unit tests with unit meist. In *Proceedings of the 10th European Software Engineering Conference Joint with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2005.
- [18] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, 2006.
- [19] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.