A String Constraint Solver for Detecting Web Application Vulnerability

Xiang Fu
Hofstra University
cscxzf@hofstra.edu

Chung-Chih Li
Illinois State University
cli2@ilstu.edu

Abstract

Given the bytecode of a software system, is it possible to automatically generate attack signatures that reveal its vulnerabilities? A natural solution would be symbolically executing the target system and constructing constraints for matching path conditions and attack patterns. Clearly, the constraint solving technique is the key to the above research. This paper presents Simple Linear String Equation (SISE), a formalism for specifying constraints on strings. SISE uses finite state transducers to precisely model various regular replacement operations, which makes it applicable for analyzing text processing programs such as web applications. We present a recursive algorithm that computes the solution pool of a SISE. Given the solution pool, a concrete variable solution can be generated. The algorithm is implemented in a Java constraint solver called SUSHI, which is applied to security analysis of web applications.

1 Introduction

Defects in user input validation are usually the cause of the ever increasing attacks on web applications and other software systems. In practice, it is interesting to automatically discover these defects and show to software designers, step by step, how the security holes lead to attacks.

In our previous work [6], we proposed a unified symbolic execution framework (as shown in Figure 1) for tackling the above challenge. Given the bytecode of a target system (e.g., a web application) and a collection of attack patterns, the framework starts with instrumenting the bytecode to prepare the system for symbolic execution [14]. Then the target system is executed as usual except that program inputs are treated as symbolic literals. Path conditions, constraints built upon symbolic literals, are used to trace the execution. A path condition records the conditions to be met by the initial values of the program input, so that the program will execute to a location. At critical points, e.g., where a SQL query is submitted, path conditions are paired with attack patterns. Solving these equations leads to attack signatures and error traces that reveal vulnerabilities.

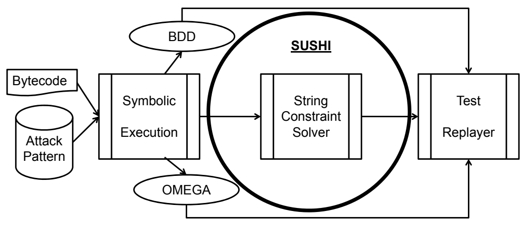

Figure 1. Unified Symbolic Framework for Vulnerability Detection

This paper presents the core algorithms of SUSHI, a constraint solver, which is the key to the aforementioned research. We study a formalism called Simple Linear String Equation (SISE), for representing path conditions and attack patterns. We present a recursive algorithm which can solve SISE constraints effectively. Intuitively, a SISE equation can be regarded as a variation of the word equation problem [17]. It is composed of word literals, string variables, and various frequently seen string operators such as substring, concatenation, and regular substitution. To solve SISE, an automata based approach is taken, where a SISE constraint is broken down into a number of *atomic* string operations. Then the solution process consists of a number of *backward image computation* steps. We show that SISE can be used to discover deeply hidden SQL injection and Cross-Site Scripting vulnerabilities. They are "corner cases" of the insufficient user input sanitation procedures, which are often neglected by security inspection tools.

The rest of the paper is organized as follows. §2 formalizes the notion of string equation system. §3 briefly describes modeling of various regular replacement semantics using finite state transducer. §4 presents a recursive algorithm for computing the solution pool of a SISE equation and analyzes the complexity of the algorithm. §5 introduces tool support. §6 presents several case study examples of SUSHI. §7 discusses related work, and §8 concludes.

2 String Equation System

This section formalizes the notion of SISE. Let N denote the set of natural numbers and Σ a finite alphabet. If $\omega \in \Sigma^*$, we say that ω is a word. We use ϵ to represent an empty word. Let R be the set of regular expressions over Σ. If $r \in R$, let $L(r)$ be the language represented by r.

2.1 String Operators

The string equation framework supports a set of five essential operators for static program analysis. The set of operators is denoted using $O = \{\circ, [i,j], x_{r\to\omega}, x^-_{r\to\omega}, x^+_{r\to\omega}\}$. They are concatenation ($\circ$), substring ($[i,j]$), declarative regular replacement ($x_{r\to\omega}$), reluctant regular replacement ($x^-_{r\to\omega}$), and greedy regular replacement ($x^+_{r\to\omega}$).

Regular replacement has several different semantics. Let $s, \omega \in \Sigma^*$ and $r \in R$. The declarative replacement, i.e., $s_{r\to\omega}$, denotes the set of all possible strings that can be obtained from s by substituting ω for every occurrence of a substring that matches $L(r)$. Notice that one declarative replacement might result in multiple words as results. The procedural replacement, i.e., the greedy and reluctant, will be different. Both of them use *left-most* matching. The greedy replacement $s^+_{r\to\omega}$ tries to match the longest string and the reluctant tries to match the shortest. The following example demonstrates their difference.

Example 2.1 Consider the following two cases. (i) If $s = aaab, r = (aa|ab)$, and $\omega = c$, then $s_{r\to\omega} = \{cc, acb\}$, and $s^-_{r\to\omega} = s^+_{r\to\omega} = cc$. (ii) If $s = aaa$, $r = a^+$, and $\omega = b$, then $s_{r\to\omega} = \{b, bb, bbb\}$, $s^-_{r\to\omega} = bbb$, and $s^+_{r\to\omega} = b$. □

We assume that there are infinitely many distinguishable string variables and let this set of variables be denoted by V. Intuitively, a *string expression* is a regular expression over Σ with occurrences of variables in V and operators in O (such as $\circ$, $[i,j]$, and $x_{r\to\omega}$). A string equation is composed of two string expressions.

Definition 2.2 Let E be the set of string expressions. A string equation is denoted by $\mu \equiv \nu$ with $\mu, \nu \in E$. A string equation system is a conjunction of a finite set of string equations. □

2.2 Simple Linear String Equation

The purpose of SISE is to capture the constraints that arise from a symbolic execution. This naturally leads to a *restricted* form of string equation, i.e., all string variables appear only on the LHS (left hand side), and the RHS (right hand side) is a regular expression.

Definition 2.3 A Simple Linear String Equation (SISE) $\mu \equiv r$ is a string equation such that $\mu \in E$, $r \in R$ provided that every string variable occurs at most once in μ. □

Definition 2.4 A solution to a SISE equation is a mapping $\rho : V \to R$ which makes the LHS equivalent to RHS (as regular expression). Let $\mu \equiv r$ be a SISE and suppose string variable v occurs in μ. The *solution pool* for v, denoted by $sp(v)$, is defined as $sp(v) = \{\omega \mid \omega \in r_2$ and $\rho(v) = r_2$ where ρ is a solution to $\mu \equiv r\}$ □

It is shown later that $sp(v)$ is a regular language for any SISE. In §4, we will describe an algorithm that takes a SISE as input and constructs as output regular expressions that represent the solution pools for all string variables in the equation.

3 Modeling Regular Replacement

Regular replacement is widely used by web application designers for sanitizing user input. This section introduces a finite state transducer model for handling various string replacement semantics. It is necessary because ignoring the difference of regular replacement semantics leads to imprecise analysis.

Consider a PHP snippet called "`postMessage`" in Listing 1. The servlet takes a message from an anonymous user and posts it on a bulletin. To prevent Cross-Site Scripting attack, the programmer calls `preg_replace()` to remove any pair of `<script>` and `</script>` tags and the contents between them. Notice that the wild card operator `*?` is a *reluctant* operator, i.e., it matches the shortest string possible. For example, given word `<script>a</script></script>`, the call on line 3 returns `</script>`. If `*` (the greedy operator) is used, the call on line 3 returns an empty string.

```
1 <?php
2   $msg = $_POST["msg"];
3   $sanitized =  preg_replace(
4         "/\<script.*?\>.*?\<\/script.*?\>/i",
5         "", $a);
6   save_to_db($sanitized)
7 ?>
```

Listing 1. Vulnerable XSS Sanitation

SUSHI is able to generate the following attack signature. Readers can verify that it is indeed effective.

```
<<script></script>script>alert('a')</script>
```

Now, a natural question following the above analysis is: *If we approximate the reluctant semantics using the greedy semantics, could the static analysis be still effective?* The

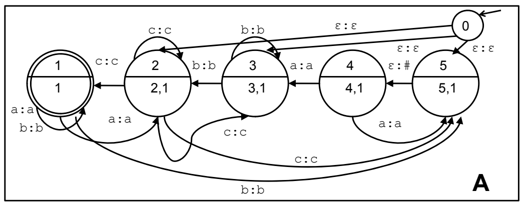

Figure 2. FST for Inserting Begin Markers

answer is clearly negative: When the `*?` operators in Listing 1 are treated as `*`, SUSHI reports no solution for the aforementioned SISE equation, i.e., a false negative report on the actually vulnerable sanitation. Thus, a precise modeling of the various regular replacement semantics is necessary.

Finite state transducer was used for processing phonological rules in [12]. We found it useful for modeling string replacements. Due to the space limit, in this section we summarize our findings and more technical details can be found in [5].

Definition 3.1 A finite state transducer (FST) is a quintuple $(\Sigma, Q, q_0, F, \delta)$ where Σ is the alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is the transition function, and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times Q$. $\square$

Note that an FST accepts a regular relation (a subset of $\Sigma^* \times \Sigma^*$), just like a Finite State Automaton (FSA) accepts a regular language. Given two FST's M_1 and M_2, let $M_1 || M_2$ denote an FST such that, $L(M_1 || M_2) = \{(s, \omega) \mid (s, \eta) \in L(M_1) \text{ and } (\eta, \omega) \in L(M_2) \text{ for some } \eta\}$. Intuitively, $M_1 || M_2$ pipes the contents of the second tape of M_1 into the first tape of M_2, and simulates M_1 and M_2 in parallel.

The key to modeling the greedy and reluctant semantics is to capture the left-most matching. Let $\# \notin \Sigma$ be a special "begin marker", and $\$ \notin \Sigma$ be the special "end marker".

A begin marker insertion performer, $\mathcal{A}$ (an FST on alphabet $\Sigma \cup \{\#\}$), can be constructed for marking the beginning of pattern $r \in R$ in any words $s \in \Sigma^*$. For example, the FST $\mathcal{A}$ in Figure 2 marks the beginning of regular pattern a^+b^+c. For instance, $(aabbcc, \#a\#abbcc) \in L(\mathcal{A})$.

With begin marker insertion performer $\mathcal{A}$, the reluctant semantics can be modeled by piping $\mathcal{A}$ with another transducer $\mathcal{A}_2$, which, whenever sees a begin marker $\#$, enters the state of conducting the replacement, and whenever a regular pattern r is matched, enters immediately the status waiting for $\#$. $\mathcal{A} \mid\mid \mathcal{A}_2$ allows to precisely model the pure reluctant semantics. The modeling of greedy semantics relies more on the power of nondeterminism and needs the composition of several more transducers. First, an end marker insertion performer nondeterministically inserts an end marker after each match of the pattern r in the input word. Then a number of filter transducers are composed to identify the "longest" match and remove extra markers. The construction details can be found in [5].

The following lemma summarizes our finding: for each of the regular replacement operators, there exists a corresponding FST. Let them be $\mathcal{M}_{r\to\omega}$, $\mathcal{M}^-_{r\to\omega}$, and $\mathcal{M}^+_{r\to\omega}$.

Lemma 3.2 For any $r \in R$ and $\omega \in \Sigma^*$ the following three finite state transducers $\mathcal{M}_{r\to\omega}$, $\mathcal{M}^-_{r\to\omega}$, $\mathcal{M}^+_{r\to\omega}$ can be constructed s.t. for any $s, \eta \in \Sigma^*$ (i) $(s, \eta) \in L(\mathcal{M}_{r\to\omega})$ iff $\eta \in s_{r\to\omega}$; and (ii) $(s, \eta) \in L(\mathcal{M}^-_{r\to\omega})$ iff $\eta = s^-_{r\to\omega}$; and (iii) $(s, \eta) \in L(\mathcal{M}^+_{r\to\omega})$ iff $\eta = s^+_{r\to\omega}$.

4 Constraint Solving

In this section we introduce the recursive algorithm that solves SISE constraints. The algorithm is able to find all solutions of a SISE constraint.

Notice that we distinguish the concept of "solution pool" and "concrete solution". Intuitively, for each variable, its solution pool is the set of all possible values it could take in some concrete solution. In our algorithm for solving SISE, the solution pool is computed first. Then concrete solutions are derived from the solution pool. The use of solution pool allows some interesting directions in our future work, e.g., it can be used for estimating the size of solution space, thus permitting the use of random testing for security analysis.

4.1 Atomic Cases

Solving a SISE can be reduced to solving four basic cases of string operators. The atomic case is trivial. That is, for SISE $E \equiv r$ if $E = x$ and $x \in V$, then the solution pool of x is simply $L(r)$. We next consider the other three cases.

Substring case: $\mu[i, j] \equiv r$ where $\mu \in E$ and $i, j \in N$ with $i \leq j$. The following equivalence is straightforward by which we can remove a substring operator.

Equivalence 1 *For any SISE of the form $\mu[i, j] \equiv r$ where $\mu \in E$ and $i, j \in N$ with $i \leq j$, ρ is a solution to $\mu[i, j] \equiv r$ iff it is a solution to $\mu \equiv \Sigma^i r[0, j - i]\Sigma^*$.*

Concatenation case: $\mu\nu \equiv r$ where $\mu, \nu \in E$.
The solution is obvious when $\nu \in R$. Consider $xr_1 \equiv r_2$ where $x \in V$ and $r_1, r_2 \in R$. This can be easily solved using *right quotient* of regular expression [10]. By convention, the right quotient $r_2/r_1 = \{x \mid xw \in r_2 \text{ } and \text{ } w \in r_1\}$. Similarly, the *left quotient* is defined as $r_2 \backslash r_1 = \{x \mid wx \in r_2 \text{ } and \text{ } w \in r_1\}$. We know that if r_1 and r_2 are regular, so are r_2/r_1 and $r_2 \backslash r_1$. The algorithm for computing regular quotient is standard.

Equivalence 2 *For any SISE of the form $\mu r_1 \equiv r_2$ ($r_1\mu \equiv r_2$) where $\mu \in E$ and $r_1, r_2 \in R$, ρ is a solution to $\mu r_1 \equiv r_2$ ($r_1\mu \equiv r_2$) iff ρ is a solution to $\mu \equiv r_2/r_1$ ($\mu \equiv r_2 \backslash r_1$).*

Now consider the general case $\mu\nu \equiv r$ where both μ and ν are non-trivial string expressions. Let $approx(\nu)$ be the result of replacing every variable in ν with Σ^*. Clearly, $approx(\nu)$ is a regular expression. We then have the following.

Equivalence 3 *For any SISE of the form $\mu\nu \equiv r_2$, ρ is a solution to $\mu\nu \equiv r_2$ iff ρ is a solution to $\mu \circ approx(\nu) \equiv r_2$.*

The solution to ν can be computed similarly. The proof can be based on the fact that a string variable cannot appear in both μ and ν.

Replacement case: $\mu_{r_1 \to \omega} \equiv r_2$ ($\mu^-_{r_1 \to \omega} \equiv r_2$, $\mu^+_{r_1 \to \omega} \equiv r_2$) where $\mu \in E$, $r_1, r_2 \in R$, and $\omega \in \Sigma^*$.
In the following we discuss the solution to $\mu_{r_1 \to \omega} \equiv r_2$. The solution to procedural replacements will be similar, because all of them use finite state transducer algorithms.

Clearly, a possible solution to $x_{r_1 \to \omega} = r_2$ is a word s such that $s_{r_1 \to \omega} \subseteq L(r_2)$. Thus, $sp(x) = \{s \mid s_{r_1 \to \omega} \subseteq L(r_2)\}$. Our goal is to construct an FST that accepts only (s, η) such that $\eta \in L(r_2)$ and η is obtained from s by replacing every occurrence of patterns in r_1 with ω. In other words, we want an FST, denoted by $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$ s.t.

$$(s, \eta) \in L(\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}) \Leftrightarrow \eta \in L(r_2) \text{ and } \eta \in s_{r_1 \to \omega}$$

We now construct $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$. Let $\mathcal{M}_1$ be the FST that accepts the identity relation $\{(s, s) \mid s \in L(r_2)\}$. Let $\mathcal{M}_{r_1 \to \omega}$ be the FST defined in Lemma 3.2, i.e., $(s, \eta) \in L(\mathcal{M}_{r \to \omega})$ iff $\eta \in s_{r_1 \to \omega}$. $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$ can then be constructed as $\mathcal{M}_{r \to \omega} || \mathcal{M}_1$. Similarly, $\mathcal{M}^-_{r_1 \to \omega \Rightarrow r_2}$ and $\mathcal{M}^+_{r_1 \to \omega \Rightarrow r_2}$ can be constructed for the pure reluctant and greedy semantics, respectively.

Equivalence 4 *For any SISE of the form $\mu_{r_1 \to \omega} \equiv r_2$ where $\mu \in E$, $r_1, r_2 \in R$, and $\omega \in \Sigma^*$. ρ is a solution to $\mu_{r_1 \to \omega} \equiv r_2$ iff it is a solution to $\mu \equiv r$ where $L(r) = \{s \mid (s, \eta) \in L(\mathcal{M}_{r_1 \to \omega \Rightarrow r_2})\}$.*

Clearly, $L(r)$ can be easily computed by projecting the FST $\mathcal{M}_{r_1 \to x \Rightarrow r_2}$ to its input tape, which results in a finite state machine, representing a regular language. The same applies to the pure greedy and reluctant semantics, using $\mathcal{M}^+_{r_1 \to \omega \Rightarrow r_2}$ and $\mathcal{M}^-_{r_1 \to \omega \Rightarrow r_2}$. Consequently, we have the following.

Theorem 4.1 Given SISE $\mu \equiv r$, for any variable v in μ, its solution pool $sp(v)$ is a regular language.

```
function solve($\mu \equiv r$)
  switch ($\mu$):
    case $x \in V$: return $\{(x, r)\}$
    case $r_1 \in R$: if $L(r_1) \cap L(r) \neq \emptyset$ return $\emptyset$ o.t. return $\perp$
    case $\mu[i, j]$: return solve($\mu \equiv \Sigma^i r[0, j - i]\Sigma^*$)
    case $\mu_{r_1 \to \omega}$: return solve($\mu \equiv \{s \mid (s, \eta) \in L(\mathcal{M}_{r_1 \to \omega \Rightarrow r})\}$)
    case $\mu^+_{r_1 \to \omega}$: return solve($\mu \equiv \{s \mid (s, \eta) \in L(\mathcal{M}^+_{r_1 \to \omega \Rightarrow r})\}$)
    case $\mu^-_{r_1 \to \omega}$: return solve($\mu \equiv \{s \mid (s, \eta) \in L(\mathcal{M}^-_{r_1 \to \omega \Rightarrow r})\}$)
    case $\mu\nu$:
      Let $r_1$ be $approx(\mu)$ and $r_2$ be $approx(\nu)$
      return solve($\mu \equiv r/r_2$) $\cup$ solve($\nu \equiv r \backslash r_1$)
```

Figure 3. Computing Solution Pool

4.2 Recursive Algorithm

Based on Equivalences 1 to 4, and Theorem 4.1, we can develop a recursive algorithm for generating the solution pool for all variables in a SISE. The algorithm is shown in Figure 3. Function `solve()` returns a set of tuples with each tuple representing a solution pool (note: not solution) for a variable. We use $\perp$ to represent "no solution". When applying any set operation (e.g., intersection and union) on $\perp$, the result is $\perp$. Note $\perp$ is not the same as empty set $\emptyset$.

Theorem 4.2 The worst complexity of the algorithm in Figure 3 is $O(|\mu| \times 2^{6 \times 2^{|\mu|} + |r|})$.

The complexity analysis needs the detailed transducer composition algorithm in [5]. We briefly discuss the sketch here. In Figure 3 the most expensive computation is the case $\mu^+_{r_1 \to \omega}$, which needs $\mathcal{M}^+_{r_1 \to \omega}$, a composition of six transducers. Among them, the size of the largest (i.e., the number of transitions plus states) is $2^{2^{|r_1|}}$ where r_1 is the regular pattern to search. Since $|r_1| < |\mu|$, we can approximate the worst complexity of handling the $\mu^+_{r_1 \to \omega}$ case as $O(2^{6 \times 2^{|\mu|} + |r|})$, because we need an additional composition operation on $\mathcal{M}^+_{r_1 \to \omega}$ with the identity relation of r. Finally, the upper bound of the recursion depth of the algorithm in Figure 3 is $|\mu|$, because there are at most $|\mu|$ operators in the LHS. This eventually leads to the complexity in Theorem 4.2.

Given the solution pool, a concrete solution can be generated by concretizing the valuation of a variable one by one using a counter loop on the number of variables in the equation. In each iteration, nondeterministically instantiate one variable from a value contained in its solution pool. Thus a new SISE equation is obtained. Solving this equation would lead to the solution pool to be used in the next iteration. Starting from the initial solution pool, the concretization process will always terminate with a concrete solution generated.

5 SUSHI Constraint Solver

SISE constraint solving is implemented in a Java library called SUSHI. This section presents some details of the tool implementation and discusses SUSHI's efficiency as a constraint solver. SUSHI includes a self-made package for supporting FST operations, and it relies on `dk.bricks.automaton` package [18] for manipulating FSA. In practice, to perform inspection on user input, FST has to handle a large alphabet represented using 16-bit Unicode. This is handled by a compression approach called SUSHI FST Transition Set. Special algorithms are developed for finite state transducer operations on SFTS. Details are not presented here due to space limit.

5.1 Evaluation of Efficiency

We are interested in the performance of SUSHI as a constraint solver. In Figure 4 we list five SISE equations for stress-testing the SUSHI package and the running statistics. Note that each equation is parametrized by an integer n (ranging from 1 to 50). For example, when n is 50, the size of the RHS of `eq4` is 100.

ID	Equation
eq1	$x \circ \mathtt{a}\{n,n\} \;\equiv\; (\mathtt{a}\vert\mathtt{b})\{2n,2n\}$
eq2	$x[n,2n] \;\equiv\; \mathtt{a}\{n,n\}$
eq3	$x \circ \mathtt{a} \circ y[0,n] \;\equiv\; \mathtt{b}\{n,n\}\mathtt{ab}\{n,n\}$
eq4	$x^{+}_{\mathtt{a}+\rightarrow\mathtt{b}\{n,n\}} \;\equiv\; \mathtt{b}\{2n,2n\}$
eq5	$\mathtt{uname='} \circ x^{-}_{\mathtt{'}\rightarrow\mathtt{''}}[0,n] \circ \mathtt{'\ pwd='} \;\equiv\; \mathtt{uname='[\hat{}'\vert'']*'}$

Figure 4. Sample Benchmark Equations

Clearly, the sample set covers all string operations we discussed earlier. In Figure 5, the first two diagrams present the size of the FST (the number of states and the number of transitions) used in the solution process, the third diagram is the size of the FSA used for representing solution pools, and the last shows the time spent on running each test. As shown in Figure 5, SUSHI scales well in most cases. The figure also suggests that the solution cost of a SISE equation mainly depends on the complexity of the automata structure of the resulting solution pool (e.g., readers can compare the cost of `eq4` and `eq5`). In addition, the experimental data (see Figures 5(a) and 5(b)) clearly indicate that to model greedy regular replacement (e.g., `eq4`) is expensive because the modeling process involves composition of six transducers.

6 Application Examples

This section presents two application examples of how SUSHI constraint solver can be used for discovering attack signatures of web application vulnerabilities.

Notice that the in-depth discussion of symbolic execution is out of the scope of this paper. Without the loss of generality, we assume that SISE constraints can be constructed by standard symbolic execution technique. In an-

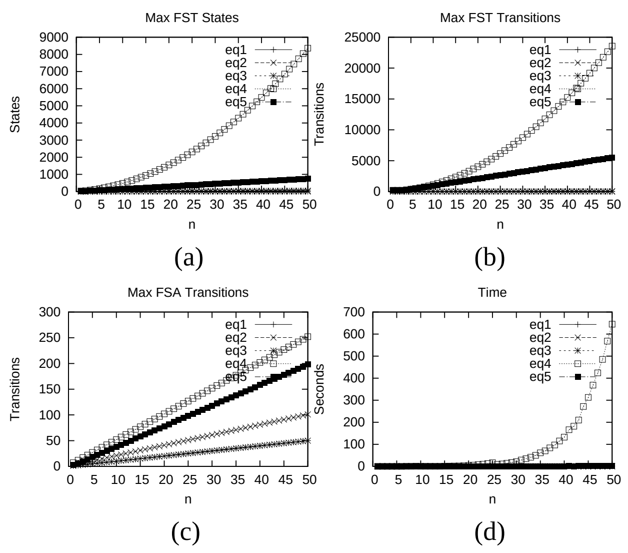

Figure 5. Constraint Solving Cost

other word, the application of SISE and its solver SUSHI library is not restricted to a certain programming language. In fact, for both of the application examples below, the SISE constraints are constructed manually. The second stage of the security analysis, i.e., attack signature generation, is automated using SUSHI. The whole process could be automated if its integration with symbolic execution engine is ready.

6.1 Discovering Password Bypassing

Consider a Java servlet called `BadLogin`, adapted from the example introduced in [6]. As shown in Listing 2, the user authentication is performed by a function `processRequest`, which reads the user input (a user name and a password), and verifies their existence in the back-end database.

To avoid attacks like SQL injection, a `massage` function is defined for applying a number of sanitation procedures. For each user input, it replaces the single quote character (a special control character in SQL) with its escaping form (a sequence of two single quotes). In addition, it limits the size of each user input string to 16 characters. Note that here 16 can be any positive integer.

The length restriction protection intends to limit the room of attackers for playing tricks. *Good intention, however, may not eventually lead to desired effects!* Combined with the string substitution, it actually causes a delicate vulnerability and the following is the *shortest* attack signature discovered by SUSHI. Notice that the pair of strings are shorter than the one originally given in [6], because the constraint solving algorithm in this paper is able to find all possible solutions.

```
sUname  =  a''''''''
sPwd    =  '' OR uname<>'
```

```
1  protected void processRequest(
2   HttpServletRequest request ...)
3   throws ServletException{
4   PrintWriter out = response.getWriter();
5   try {
6      String sUname = request.getParameter("sUname");
7      String sPwd = request.getParameter("sPwd");
8      Connection con = DriverManager.getConnection("...");
9      Statement stmt = con.createStatement();
10     String strCmd= "SELECT␣*␣FROM␣users\nWHERE␣uname='"
11       + massage(sUname) + "'␣AND␣pwd='"
12       + massage(sPwd) + "'";
13     ResultSet srs = stmt.executeQuery(strCmd);
14     if(srs.next()){
15      out.println("Welcome␣" + sUname);
16     }else{
17      out.println("Login␣fail!");
18     }
19  }catch(Exception exc){...}
20 }
21
22 protected String massage(String str){
23  String strOut = str.replaceAll("'", "''");
24  if(strOut.length()>16) return strOut.substring(0,16);
25  return strOut;
26 }
```

Listing 2. Vulnerable Authentication

The trick is explained briefly as below. By applying the `massage()` function on `sUname`, each of the 8 single quotes is converted to a sequence of two quotes. However, the last quote is chopped off by the `substring()` function (at the 16'th character as shown in Listing 2). This results in the following SQL query, which bypasses password checking. Notice that the logical structure of the `WHERE` clause has been changed by the attack string, because the single quotes are treated differently – each pair of single quotes between `a` and `AND` is regarded as the escaping form of one single quote by SQL parser.

```
SELECT uname, pwd FROM users
WHERE uname='a''''''''''''''''␣AND␣pwd=''''' OR uname<>''
```

Clearly, the vulnerability discussed in this section cannot be discovered by black-box testing like [4, 11, 20], because without prior knowledge of the implementation, it is very hard to craft the input strings that could pass the sanity check by `massage()`. The string analyses such as [1, 3, 13] cannot discover the bug either, because regular replacement is not supported. Only when the control/data flow information is taken advantage by the automated vulnerability scanner, such deeply hidden bugs can be revealed.

Detection using SUSHI: We now briefly describe how the aforementioned attack could be discovered by SUSHI. When `processRequest` is symbolically executed, the value of variables `sUname` and `sPwd` should be initialized with symbolic literals and let them be x and y. Then, by executing the statements one by one, at line number 13 where the SQL query is submitted, the symbolic value of `strCmd` is a concatenation of the following string terms.

1. constant word `SELECT * FROM users \nWHERE uname='`
2. term $x^{+}_{'\to''}[0, 16]$
3. constant word `' AND pwd='`
4. term $y^{+}_{'\to''}[0, 16]$
5. constant word `'`

Let the string expression (concatenation of the above terms) be S_1. Here $x^{+}_{'\to''}[0, 16]$ represents the output of the `massage()` function on `sUname`, i.e., to replace every single quote with its escaping form and then to perform a substring operation on the user input. Similar is $y^{+}_{'\to''}[0, 16]$. Now by associating the symbolic string expression with predefined attack patterns, we can construct SISE equations. For example, the following is a sample SISE equation, based on one of the pre-collected SQL injection attack patterns:

$$S_1 \equiv \texttt{uname='([\^{}']|'')}^{*}\texttt{' OR }^{*}\texttt{uname<>''}$$

Intuitively, the SISE equation asks the following question: after all the sanitation procedures are applied, is it feasible to make the `WHERE` clause of the SQL query essentially a tautology (by "`OR uname<>''`")? Using SUSHI, we are able to generate the *shortest* attack strings in 1.4 seconds. The detailed information is shown in Figure 6. The first four columns show the size of FST and FSA used in solving the SISE equation. The last column shows the overall time cost.

6.2 Generating Shortest XSS Exploits

We demonstrate how *reusable* attack pattern rules are represented in SISE. We show how SUSHI is used for analyzing one recently discovered XSS vulnerability [16] in Adobe Flex SDK 3.3. A file named `index.template.html` is used for generating wrappers of application files in a FLEX project. It takes a user input in the form of "`window.location`" (URL of the web page being displayed), which is written into the DOM structure of the HTML file using `document.write(str)`.

Clearly, the unfiltered input could lead to XSS (a tainted analysis [19] could identify the existence of the vulnerability). However, to precisely craft a working exploit is still not a trivial work, as several constraints have to be satisfied before the injected JavaScript code could work. For example, the injected JavaScript tag should not be contained in the value of an HTML attribute (otherwise, it will not be executed). In addition, the resulting HTML should remain syntactically correct, at least until the parser reaches the injected JavaScript code.

Example	FST_States	FST_Trans	FSA_States	FSA_Trans	Time
Bypass	238	1634	17	62	1.4s
XSS	0	0	272	4217	74.1s

Figure 6. Attack Signature Generation Cost

Rule	Regular Expression Pattern
XSS	`.*<script>alert('XSS found!')</script>.*`
EffectiveScript	$\overline{\texttt{.*[a-zA-Z0-9\_]+ *= *"[\^{}"]*<script.*>.*}}$
MatchTag	$\overline{\texttt{.*<embed[\^{}<>]*>.*} \cap \overline{\texttt{.*</embed[\^{}<>]*>.*}}}$

Figure 7. Rules in RHS

SUSHI can help generating the attack string precisely. In fact, SUSHI generates the following attack string which is, first of all *working*, and is *shorter* (if not the shortest) than the exploit given in the original securitytracker post [16].

```
\"<script>alert('XSS found!')</script>
```

In the following, we briefly describe how the SISE equation is constructed for generating the exploit. The LHS (left hand side) of the equation is a concatenation of three strings, two constant words and one variable. The variable represents the unsanitized user input. The two constant words represent the other parts of the parameters collected and combined by the vulnerable JavaScript code snippet. The two constant words are obtained by manual analysis that simulates symbolic execution on the vulnerable JavaScript snippet. The size of the constant words in LHS (combined) is 445 characters long. The RHS is a conjunction of a number of attack patterns and filter rules as shown in Figure 7. Notice that these attack patterns, apparently, are *reusable*.

The `XSS` pattern is straightforward. It requires that the JavaScript `alert()` function eventually shows up in the combined output. Then the `EffectiveScript` rule forbids the JavaScript snippet to be embedded in any HTML attribute definition (thus ineffective). The `MatchTag` rule requires that an HTML beginning tag must be matched by an ending tag (in our case the "`<embed>`" tag). Clearly, the above rules are general and can be applied to analyzing other XSS attacks.

The cost of finding the shortest solution is shown in Figure 6. Notice that the solution cost is expensive because the equation constructed is unusually large (the size of LHS is 445). This is reasonable compared with similar efforts in the area, e.g., HAMPI [13].

7 Related Work

String analysis, i.e., analyzing the set of strings that could be produced by a program, emerged as a novel technique for analyzing web applications, e.g., compatibility check of XHTML files against schema [2], security vulnerability scanning [6, 7], and web application verification [1]. Solving string constraints is one of the many directions for tackling command injection attacks (e.g., tainted analysis [19], forward string analysis [3], run-time hardening [8]), black-box testing [11]). While the aforementioned techniques are mainly *dynamic* analysis, string analysis is mostly static. Static analysis helps to discover vulnerabilities before web applications are deployed.

In general, there are two interesting directions of string analysis: (1) *forward analysis*, which computes the image (or its approximation) of the program states as constraints on strings and other primitive data types; and (2) *backward analysis*, which usually starts from the negation of a property and computes backward. Most of the related work (e.g., [1–3, 15]) fall into the category of forward analysis. The work presented in this paper is *backward*. Compared with forward string analysis, the backward string analysis is able to generate attack strings as hard evidence of a vulnerability and avoids *false positives*. However, it suffers from false negatives, i.e., there are cases where vulnerabilities are ignored by the analysis.

SISE can be regarded as a variation of the *word equation* problem [17]. Note that in a word equation, only word concatenation is allowed. In SISE, various popular `java.regex` operations are supported.

SISE is a continuation of our earlier efforts of building a unified symbolic execution framework [6]. The SISE framework subsumes the string constraint framework outlined in [6], in which a number of incomplete heuristics algorithms are used for handling string concatenation and constant string replacement, suffering from imprecision. This paper provides a both sound and complete algorithm, which is able to produce all possible solutions of a SISE constraint. As shown in §3 of this paper, the modeling of various regular replacement semantics largely improves the analysis precision. The SUSHI constraint solver is implemented and is applied to a number of case study examples, which allows us to verify the effectiveness of the proposed string analysis in practice.

The closest work to ours is probably the HAMPI string constraint solver [13], which is also a backward analysis. HAMPI supports solving string constraints with context-free components, which are essentially unfolded to regular language within a bound. HAMPI, however, does not support string replacement nor regular replacement, which significantly limits its ability to reason about sanitation procedures. In addition, the unfolding of context-free components limits its scalability. Similarly, Hooimeijer and Weimer's work [9] in the decision procedure for regular constraints does not support regular replacement. Another close work to ours is Yu's automata based forward/backward string analysis [22]. Yu *et al.* use language based replacement [23] to handle regular replacement. Imprecision is introduced in the over-approximation during the language based replacement. Conversely, our analysis considers the delicate differences among the typical regular replacement semantics.

In [21] Wassermann and Su combined string analysis and taint analysis for discovering Cross-Site Scripting attacks. Their analysis is forward and does not directly generate attack signatures. Finite state transducer is used in [21] for modeling regular replacements, however, it does not distinguish between the various semantics such as the greedy and the reluctant, which is addressed by this paper.

8 Conclusion

This paper introduces a string constraint solver called SUSHI. We show that a fragment of the general string equation problem, called Simple Linear String Equation (SISE), can be solved using automata based approach. Finite state transducer is used for precisely modeling several different semantics of regular substitution. This helps to reduce false positives of a string analysis. The SUSHI constraint solver is implemented and it is applied to analyzing security of several small web applications. The experimental results show that SUSHI works efficiently in practice. We plan to integrate SUSHI with existing symbolic execution engines to automate vulnerability exploration. Future directions include expanding the solver to consider context-free components, and incorporating temporal logic operators for modeling malicious attack behaviors. We are also interested in applying the work to automated grading in computer science education.

Acknowledgment: this work is partially supported by NSF under contract DUE-0836859 and DUE-0837020.

References

[1] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools AND Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 322–336. Springer, 2009.

[2] A. S. Christensen, A. Møler, and M. I. Schwartzbach. Extending java for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

[3] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[4] CIRT INC. Nikto. *available at* `http://www.cirt.net/nikto2`.

[5] X. Fu and C. Li. Modeling regular replacement for string constraint solving. In *Proc. of the 2nd NASA Formal Methods Symposium*, pages 67–76, 2010.

[6] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. In *Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 87 – 96, 2007.

[7] 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.

[8] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pages 174–183, 2005.

[9] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languagees. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, 2009.

[10] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[11] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 11th International World Wide Web Conference (WWW 2003)*, 2003.

[12] R. M. Kaplan and M. Kay. Regular models of phonological rule systems. *Computational Linguistic*, 20(3):331–378, 1994.

[13] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for string constraints. In *Proc. of 2009 International Symposium on Testing and Analysis (ISSTA'09)*, 2009.

[14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[15] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*, August 2006.

[16] labs@gdssecurity.com. Adobe flex sdk input validation bug in 'index.template.html' permits cross-site scripting attacks. `http://www.securitytracker.com/alerts/2009/Aug/1022748.html`, 2009.

[17] M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.

[18] A. Møller. The dk.brics.automaton package. *available at* `http://www.brics.dk/automaton/`.

[19] 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.

[20] SPI Dynamics. Webinspect: Security throughout the application lifecycle. Datasheet. `http://www.spidynamics.com/assets/documents/WebInspect_DataSheets.pdf`.

[21] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. of the 30th International Conference on Software Engineering*, pages 171–180, 2008.

[22] F. Yu, M. Alkhalaf, and T. Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009.

[23] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc. of the 15th SPIN Workshop on Model Checking Software*, pages 306–324, 2008.