On Simple Linear String Equations

Xiang Fu[1] and Chung-Chih Li[2] and Kai Qian[3]

[1] Hofstra University, Xiang.Fu@hofstra.edu
[2] Illinois State University, cli2@ilstu.edu
[3] Southern Polytechnic State University, kqian@spsu.edu

Abstract. This paper presents a novel backward constraint solving technique for analyzing text processing programs. String constraints are represented using a variation of word equation called Simple Linear String Equation (SLSE). SLSE supports precise modeling of various regular string substitution semantics in Java regex, which allows it to capture user input validation operations widely used in web applications. On the other hand, SLSE is more restrictive than a word equation in that the location where a variable can occur is restricted. We present the theory of SLSE, and a recursive algorithm that can compute the solution pool of an SLSE. Given the solution pool, any concrete variable solution can be generated. The algorithm is implemented in a Java library called SUSHI. SUSHI can be applied to vulnerability analysis and compatibility checking. In practice, it generates command injection attack strings with very few false positives.

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. This paper is about one initial step in answering the following question:

Given the bytecode of a software system, is it possible to automatically generate attack strings that exploit the vulnerability resident in the system?

A solution to the above problem that is both sound and complete is clearly impossible, according to Rice's Theorem. However, fragments of the general problem may be decidable. This work is a part of our preliminary efforts [6, 7] for building a unified symbolic execution framework that can discoverer web application vulnerabilities automatically. At critical points during a symbolic execution [15], e.g., where a SQL query is submitted, path conditions are solved to match attack patterns. Solving these equations leads to attack strings and error traces that reveal vulnerabilities. This paper is about how to solve string constraints. We study a variation of the general word equation problem called Simple Linear String Equation (SLSE).

An automata based approach is taken to solve SLSE, where an SLSE is broken down into a number of *atomic* string operations. Then the solution process

consists of a number of *backward image computation* steps. This is quite different from, e.g., the solution of word equations using Makanin's algorithm [19], because we take advantage of the fact that each variable occurs only once in an SLSE. 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 a maximal set of strings X where for each string $s \in X : f(s) \in R$. For most string operations, backward image computation can be defined using regular expressions. Solving string substitution can be realized using finite state transducer. String concatenation operations are handled by graph search algorithm on automata. This paper has the following contributions:

1. **Simple Linear String Equation:** SLSE was informally studied as a case study in [6]. This paper formalizes the concept, and proposes a recursive algorithm that computes the solution pool, which can be used for computing any variable solution to the equation.
2. **Precise Modeling of Regular Substitution:** In [6, 7], only constant string substitution is supported. This paper provides modeling of various regular substitution semantics (e.g., greedy and reluctant) of Java regex class, which avoids false positives in practice.
3. **SUSHI library:** A string solver library called SUSHI for SLSE equations is constructed. The library can be used with symbolic execution engines, model checkers, as well as manually.

The rest of the paper is organized as follows. §C2 defines the general model of the string equation system. §C3 provides modeling of various java regex regular replacement semantics. §C4 presents a recursive algorithm for computing the solution pool of a simple linear string equation. §C5 introduces tool support and case studies of SLSE. §C6 discusses related work. §C7 concludes.

2 String Equation System

In this section we define notations and terminologies that will be used throughout the paper. Let N denote the set of natural numbers and Σ a finite alphabet. If $\omega \in \Sigma^*$, we say that ω is a word. Let R be the set of regular expressions over Σ. If $r \in R$, let $L(r)$ be the language represented by r. We abuse the notation by writing $\omega \in r$ if $\omega \in L(r)$, when the context is clear that r is a regular expression. We assume that there are infinitely many distinguishable string variables and let this set of variables be denoted by V. We use $O = \{\circ, [i,j], x_{r\to\omega}, x^{-}_{r\to\omega}, x^{+}_{r\to\omega}\}$ to represent the set of string operators (semantics defined in §C2.2).

2.1 String Equations

Intuitively, a *string expression* is a regular expression over Σ with occurrences of variables in V and $[i,j]$ (`substring`) and $x_{r\to\omega}$ (`replacement`) operators.

Definition 1. *Let E denote the set of string expressions, which is defined as follows:*

1. *If* $x \in (V \cup R)$, *then* $x \in E$.
2. *If* μ, $\nu \in E$, *then* $\mu\nu \in E$.
3. *If* $\mu \in E$, *then* $\mu[i, j] \in E$.
4. *If* $\mu \in E$, *then* $\mu_{r\to\omega}$, $\mu^{-}_{r\to\omega}$, $\mu^{+}_{r\to\omega} \in E$ *for all* $r \in R$ *and* $\omega \in \Sigma^*$.
5. *Nothing else is in* E *except those described above.*

Ambiguity can arise when infix notation is used without proper precedence, e.g., for $xy[0, 2]$ the `substring` operator can be applied to xy or y. Thus, we sometimes write a string expression μ in postfix and let it be $\mathcal{P}(\mu)$. Clearly, $\mathcal{P}(\mu) = (\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n)$ is a list of terms where for each $1 \leq i \leq n$: $\alpha_i \in V \cup R \cup O$. For example, let $\mu = xy[0, 2]$, then $\mathcal{P}(\mu) = (x, y, [0, 2], \circ)$, with $\circ$ for concatenation, represents the case with `substring` applied to y only.

By convention, given $\mu \in E$ with $\mathcal{P}(\mu) = (\alpha_1, \ldots, \alpha_m)$, and $1 \leq x_1, \ldots, x_n \leq m$ and $s_1, \ldots, s_n \in R$, we use $\mu_{(x_1/s_1, x_2/s_2, \ldots, x_n/s_n)}$ to denote the object obtained by replacing $\alpha_{x_1}, \alpha_{x_2}, \ldots, \alpha_{x_n}$ in $\mathcal{P}(\mu)$ with $s_1, s_2, \ldots, s_n$, respectively. It is required that for $1 \leq i \leq n$: $\alpha_{x_i} \in V \cup R$ (i.e., operators will not be affected). We then define several terms used for defining solutions of a string equation.

Definition 2. *A mapping* ρ *is a set of tuples* $\{(x_1, s_1), (x_2, s_2), \ldots, (x_n, s_n)\} \subseteq (V \cup R) \times R$ *that satisfies the following: (i)* ρ *is single-valued, i.e., for any* $x_i, x_j \in V$ *with* $1 \leq i, j \leq n$: $x_i = x_j$ *iff* $i = j$*; and, (ii)* ρ *is consistent, i.e., if* $x_i \in R$, *then* $L(s_i) \subseteq L(x_i)$.

In the definition above, being single-valued is a standard requirement to have the mapping function well-defined. A variable used in constructing a string equation should not be mapped to two values, however, a regular expression can. Being consistent forces the replacement not to replace a regular expression by an arbitrary string but confine the substitute within $L(r)$. Consider the following example. Let $x \in V$ and $a, b, c \in \Sigma$, then $\rho = \{(x, ab), ((abc)^*, abcabc)\}$ is a mapping by Definition 2, however, $\rho = \{(x, ab), ((abc)^*, abcab)\}$ is not because $abcab \notin L((abc)^*)$. A mapping itself cannot be used for defining a solution, as the syntax "structure" information of string expression terms has not been attached yet. For example consider a^*xa^* with $x \in V$ and $a \in \Sigma$, we need a way to distinguish the two a^*'s before and after x.

Definition 3. *Given a postfix string expression* $\mathcal{P}(\mu) = (\alpha_1, \ldots, \alpha_m)$ *and a mapping* $\rho = \{(x_1, s_1), \ldots, (x_n, s_n)\}$, *the structure labeling function that associates* $\mathcal{P}(\mu)$ *and* ρ *is a function* $\mathcal{B} : [1, m] \to [1, n] \cup \{\perp\}$ *which satisfies the following. For any* $1 \leq u \leq m$: *if* $\mathcal{B}(u) \neq \perp$ *then* $\alpha_u = x_{\mathcal{B}(u)}$. *Then given* ρ *and* $\mathcal{B}$, *let* $\mathcal{D} = \{i \mid \mathcal{B}(i) \neq \perp\}$ *and* $|\mathcal{D}|$ *the size of* $\mathcal{D}$. *The valuation function* $\phi_{\rho,\mathcal{B}}(\mu)$ *is defined as*

$$\phi_{\rho,\mathcal{B}}(\mu) = \mu_{(a_1/s_{\mathcal{B}(a_1)}, a_2/s_{\mathcal{B}(a_2)}, \ldots, a_{|\mathcal{D}|}/s_{\mathcal{B}(a_{|\mathcal{D}|})})}$$

where for each $1 \leq j \leq |\mathcal{D}|$: a_j *is a distinct member of* $\mathcal{D}$.

Definition 4. *A string equation is denoted by* $\mu \equiv \nu$ *with* $\mu, \nu \in E$. *We say that* $\rho \subseteq (V \cup R) \times R$ *is a solution to the equation if there exists two structure labeling functions* $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *s.t.* $\phi_{\rho,\mathcal{B}_1}(\mu) = \phi_{\rho,\mathcal{B}_2}(\nu)$, *and for every* $(x_u, s_u) \in \rho$ *there exists* $j \in N$ *s.t.* $\mathcal{B}_1(j) = u$ *or* $\mathcal{B}_2(j) = u$.

When context is clear we use ϕ_ρ to denote a valuation function by omitting the details of structure labeling. For example, let $x, y \in V$, $a, b, c \in \Sigma$, and $\rho = \{(x, c), (ab, ab), (ca, ca), (y, b)\}$. One can verify that ρ is a solution to string equation $xab \equiv cay$, since $\phi_\rho(xab) = \phi_\rho(cay) = cab$. Notice that a string equation, when treated as a string constraint, holds as long as there is a valuation function that makes the left hand side (LHS) equivalent to the right hand side (RHS), as regular expressions. Consider the equation $a^* \equiv a^*b^*$, it has a solution $\{(a^*, a^*), (a^*b^*, a^*)\}$. We can directly extend the definition above to a string equation system that is simply a finite set of string equations. We say that ρ is a solution to a system iff ρ is a solution to every string equation in the system.

2.2 Semantics

We now define semantics of operators. Concatenation is represented using $\circ$. For $s \in \Sigma^*$, $s[i, j]$ denotes the substring of s starting from index i up to index $j - 1$ (included). If $r \in R$, then let $r[i, j] = \{s[i, j] \mid s \in r\}$. For $s, \omega \in \Sigma^*$, and $r \in R$, $s_{r\to\omega}$ denotes the set of all possible strings that can be obtained from s by substituting ω for every occurrence of a substring that matches $L(r)$. Moreover, let $s^-_{r\to\omega}$ denote the string obtained with *pure reluctant left-most pattern matching procedure*, and $s^+_{r\to\omega}$ with *pure greedy left-most pattern matching procedure*. Formally, we denote the three regular replacement operators as follows:

Definition 5. *Let $s, \omega \in \Sigma^*$ and $r \in R$ with $\epsilon \notin r$.* [4]

$$s_{r\to\omega} = \begin{cases} \{s\} & \text{if } s \notin \Sigma^* r \Sigma^*; \\ \{\nu_{r\to\omega}\omega\mu_{r\to\omega} \mid s = \nu\beta\mu, \beta \in r\} & \text{otherwise.} \end{cases}$$

If $s_{r\to\omega} = \{s\}$, then let $s^-_{r\to\omega} = s^+_{r\to\omega} = s$; otherwise, define

- $s^-_{r\to\omega} = \nu\omega\mu^-_{r\to\omega}$ *where $s = \nu\beta\mu$ such that, $\nu \notin \Sigma^* r \Sigma^*$, and $\beta \in r$, and for every x, y, u, t, m, n with $\nu = xy$, $\beta = ut$, and $\mu = mn$: if $y \neq \epsilon$ then $yu \notin r$ and $y\beta m \notin r$; and if $t \neq \epsilon$ then $u \notin r$.*
- $s^+_{r\to\omega} = \nu\omega\mu^+_{r\to\omega}$ *where $s = \nu\beta\mu$ such that, $\nu \notin \Sigma^* r \Sigma^*$, and $\beta \in r$, and for every x, y, u, t, m, n with $\nu = xy$, $\beta = ut$, and $\mu = mn$: if $y \neq \epsilon$ then $yu \notin r$ and if $m \neq \epsilon$ then $y\beta m \notin r$.*

For any $s \in \Sigma^*$, $s_{r\to\omega}$ is a set of words but $s^-_{r\to\omega}$ and $s^+_{r\to\omega}$ are words. Moreover, $s_{r\to\omega}$ is declarative while $s^-_{r\to\omega}$ and $s^+_{r\to\omega}$ are procedural as they are defined based on a reluctant and a greedy procedures, respectively, with left-most pattern matching. One can verify that $s^-_{r\to\omega}$ and $s^+_{r\to\omega}$ are uniquely defined for any $s \in \Sigma^*$ and $r \in R$ with $\epsilon \notin r$. Consider the following two examples. (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 can extend the definition of replacement to set of words as follows.

Definition 6. *Let $S \subseteq \Sigma^*$, $r \in R$ and $\omega \in \Sigma^*$, define (i) $S_{r\to\omega} = \bigcup_{s\in S} s_{r\to\omega}$, (ii) $S^-_{r\to\omega} = \{s^-_{r\to\omega} \mid s \in S\}$, and (iii) $S^+_{r\to\omega} = \{s^+_{r\to\omega} \mid s \in S\}$.*

[4] In practice, SUSHI uses extra finite state transducer filters for handling $\epsilon \in r$. For example, if $s = a$, $r = a^*$, and $\omega = b$, then in SUSHI $s^-_{r\to\omega} = bab$, $s^+_{r\to\omega} = bb$.

3 Modeling Regular Replacement

In this section we introduce an augmented finite state transducer model for various string replacement semantics. Finite state transducer is widely used for recognizing regular relations and for processing phonological rules [13]. We found it also useful for string replacements.

Definition 7. *Let Σ^ϵ denote $\Sigma \cup \{\epsilon\}$. A finite state transducer (FST) is an enhanced two-taped nondeterministic finite state machine described by a quintuple $(\Sigma, Q, q_0, F, \delta)$, where Σ is the alphabet, Q the set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ the set of final states, and δ is the transition function, which is a total function of type $Q \times \Sigma^\epsilon \times \Sigma^\epsilon \to 2^Q$.*

By convention, let the second and third arguments of the transition function come from the current symbols on first and second tapes, respectively. It is easy to argue that with an appropriate coding method, adding an additional input tape to the standard finite state machine does not enhance the power of a finite state machine. What makes FST more powerful is to allow a transition based on only one symbol from either one of the two tapes. For example, $q_j \in (q_i, a, \epsilon)$ means that if the current symbol in the first input tape is a, the machine can transfer from state q_i to q_j without reading (i.e., consuming) the current symbol in second tape.

Let L_1 and L_2 be two languages. If an FST, M, accepts (ω_1, ω_2) iff $(\omega_1, \omega_2) \in L_1 \times L_2$, we say that M recognizes the language pair (L_1, L_2), and is denoted by $M_{L_1 \times L_2}$. It is straightforward to argue that, L_1 and L_2 are regular iff $M_{L_1 \times L_2}$ exists. For convenience, we can extend the transition labels to regular relations, obtaining an *augmented FST*, denoted by AFST.

Definition 8. *An augmented finite state transducer (AFST) is an FST $(\Sigma, Q, q_0, F, \delta)$ with the transition function augmented to type $Q \times \mathcal{R} \to 2^Q$, where $\mathcal{R}$ is the set of regular relations over Σ.*

Note that, while we have tried to keep our setup as general as possible, we would often restrict the transition function of an AFST to the following two types: (1) $Q \times R \times \Sigma^* \to 2^Q$. In a transition diagram, we label the arc from q_i to q_j for transition $q_j \in \delta(q_i, r, \omega)$ by $r : \omega$; and (2) $Q \times \{Id(r) \mid r \in R\} \to 2^Q$ where $Id(r) = \{(\omega, \omega) \mid \omega \in L(r)\}$. In a transition diagram, an arc of type (2) is labeled as $Id(r)$.

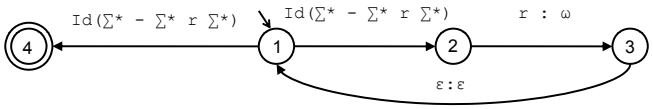


Fig. 1. An FST for $s_{r \to \omega}$

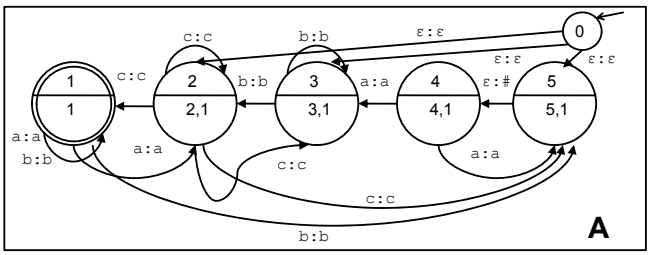


Fig. 2. An FST for Inserting Begin Markers

Now, we can use an AFST to model the declarative string replacement $s_{r\to\omega}$ for any $\omega \in \Sigma^*$ and $r \in R$ (with $\epsilon \notin r$). Figure 1 shows the construction, which presents an AFST that accepts $\{(s,\eta) \mid s \in \Sigma^*$ and $\eta \in s_{r\to\omega}\}$. In other words, given any two $s, \eta \in \Sigma^*$, we can use the AFST to check if η is a string obtained from s by replacing every occurrence of patterns in r with ω. We alternatively use FST and AFST for the time being without loss of generality since it is clear that every AFST has an corresponding FST to recognize the same language pair.

The AFST in Figure 1 uses nondeterminism to handle the declarative nature of $s_{r\to\omega}$. It is known that the nondeterministic transducer (NFST) is more powerful than the deterministic one (DFST), which differs with the well known fact that DFA and NFA are equivalent. In [9] we show that certain fragments of the general problem can be modeled using deterministic finite state transducer (DFST) under certain restrictions. Next we briefly discuss the construction of FST for modeling the procedural semantics.

3.1 Modeling Procedural Regular Replacement

Finite state transducers can be constructed for modeling both the pure reluctant and greedy semantics. We briefly describe the main idea of the construction algorithm, and the complete details can be found in our technical report [9]. We fix some notations first. Given two FST's M_1 and M_2, let $M_1||M_2$ denote a 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. Clearly, $L(M_1||M_2)$ represents the composition of two regular relations. The construction algorithm can be found in [13].

The key to modeling 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". Given a word $\omega \in (\Sigma \cup \{\#, \$\})^*$, the projection operator $\pi(\omega)$ produces a word $\omega' \in \Sigma^*$ s.t. all markers are removed from ω.

A begin marker inserter $\mathcal{A}$ (an FST on alphabet $\Sigma \cup \{\#\}$) can be constructed for marking the beginning of pattern $r \in R$ in any words $s \in \Sigma^*$ s.t. for each $(s, \eta) \in L(\mathcal{A})$ the following are satisfied: (i) $s = \pi(\eta)$; and (ii) if $\eta[i] = \#$ then $\pi(\eta[i+1, |\eta|]) \in r\Sigma^*$, and (iii) every $\#$ in η is not immediately preceded or followed by another $\#$. 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 inserter $\mathcal{A}$, 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 (note extra markers have to be removed during replacement process), and whenever a regular pattern r is matched, enters immediately (i.e., without waiting for longer matches) the status waiting for $\#$. $\mathcal{A} \,||\, \mathcal{A}_2$ 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 inserter *nondeterministically* inserts end markers after pattern r in the input word. Then a number of filter transducers are composed to identify the "longest" match and remove extra markers. We omit the details here for space limit and details of the construction algorithm can be found in our technical report [9].

Lemma 1. *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 Simple Linear String Equation

In this section we narrow down our scope to a kind of simplified liner string equations called *Simple Linear String Equations (SLSE)*. Compared to the general string equations given in Definition 1, SLSE is easy to solve and yet it suffices to formulate, for example, command injection security problems in many web applications. Our approach is to break SLSE into several basic cases and then combine their solutions to obtain the general solution. We define SLSE as follows.

Definition 9. *A Simple Linear String Equation (SLSE) $\mu \equiv r$ is a string equation such that $\mu \in E$, $r \in R$ provided that every string variable occurs at most once in μ.*

Definition 10. *Let $\mu \equiv r$ be an SLSE. We say that ρ is a variable solution to $\mu \equiv r$ iff $\rho = \tau \cap (V \times R)$ and τ is some solution to $\mu \equiv r$.*

Definition 11. *Let $\mu \equiv r$ be an SLSE and suppose string variable v occurs in μ. The* solution pool *for v, denoted by $sp(v)$, is defined as follows.*

$$sp(v) = \{\omega \mid \omega \in r_2 \text{ and } (v, r_2) \in \rho \text{ where } \rho \text{ is a variable solution to } \mu \equiv r\}$$

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

4.1 Solving Basic Case of SLSE

According to Definition 1, E is constructed recursively based on the atomic case (rule 1) and three operations: concatenation (rule 2), substring (rule 3), and string replacement (rule 4). Thus, solving an SLSE can be reduced to solving the four basic cases. The atomic case is trivial. That is, for SLSE $E \equiv r$ if $E = x$ and $x \in V$, then the solution pool of x is simply $L(r)$.

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

In the following easy example, we will see how to put Definitions 1, 2, 4, 11 into the picture. Consider SLSE $x[2,4] \equiv ab^*$ where $x \in V$ and $a, b \in \Sigma$. Using Equivalence 1 we obtain $x \equiv \Sigma^2 ab^*[0,2]\Sigma^*$ and hence $sp(x) = \Sigma^2 ab\Sigma^*$. Consider an arbitrary word in $sp(x)$, e.g., $x = ccabccc$. Let $\tau = \{(x, ccabccc), (ab^*, ab)\}$. According to Definitions 2 and 4, the mapping τ is well-defined and it is a solution to $x[2,4] \equiv ab^*$, since $\phi_\tau(x[2,4]) = ccabccc[2,4] = ab = \phi_\tau(ab^*)$. According to to Definition 10, $\rho = \tau \cap (V \times R) = \{(x, ccabccc)\}$ is a variable solution to the equation. Finally, according to Definition 11, $ccabccc \in sp(x)$.

Concatenation case: $\mu\nu \equiv r$ where $\mu, \nu \in E$. The equivalence is obvious when $\nu \in R$. Consider $xr_1 \equiv r_2$ where $x \in V$ and $r_1, r_2 \in R$. By convention, we denote the quotient of r_2 with respect to r_1 by r_2/r_1. We know that if r_1 and r_2 are regular, so is r_2/r_1. In this trivial case, $sp(x) = r_2/r_1$. Regular quotient can be easily computed using a graph search on automata, e.g., as shown in [6]. Thus, we have the following equivalence.

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

For the general concatenation case, we can reduce it to the case in Equivalence 2 by using other equivalences first.

Replacement case: $\mu_{r_1 \to \omega} \equiv r_2$ where $\mu \in E$, $r_1, r_2 \in R$, and $\omega \in \Sigma^*$. We also discuss the cases for procedural replacements. 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 shown in Figure 1, i.e., $(s, \eta) \in L(\mathcal{M}_{r \to \omega})$ iff $\eta \in s_{r_1 \to \omega}$. It is clear that $\mathcal{M}_{r_1 \to \omega \Rightarrow r_2}$ is constructed

as $\mathcal{M}_{r\to\omega}||\mathcal{M}_1$. Similarly, for the pure reluctant semantics, $\mathcal{M}^-_{r_1\to\omega\ \Rightarrow r_2}$ can be constructed as $\mathcal{M}^-_{r_1\to\omega}||\mathcal{M}_1$ (where $\mathcal{M}^-_{r_1\to\omega}$ is defined in Lemma 1). Clearly, the following is true:

$$(s,\eta) \in L(\mathcal{M}^-_{r_1\to\omega\ \Rightarrow r_2}) \Leftrightarrow \eta = s^-_{r_1\to\omega} \in L(r_2)$$

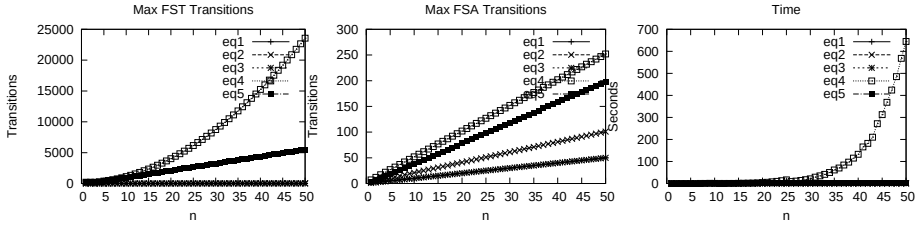
The FST $\mathcal{M}^+_{r_1\to\omega\ \Rightarrow r_2}$ can be defined similarly for the greedy semantics.

Equivalence 3 *For any SLSE 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 variable solution to $\mu_{r_1\to\omega} \equiv r_2$ iff it is a variable solution to $\mu \equiv r$ where $L(r) = \{s \mid (s,\eta) \in L(\mathcal{M}_{r_1\to x\ \Rightarrow r_2})\}$.*

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. The same applies to the pure greedy and reluctant semantics, using $\mathcal{M}^+_{r_1\to\omega\ \Rightarrow r_2}$ and $\mathcal{M}^-_{r_1\to\omega\ \Rightarrow r_2}$. Based on Equivalences 1,2, and 3, it is clear that we can develop a recursive algorithm for generating the solution pool for all variables in an SLSE. Given the solution pool, any concrete solution can be generated by concretizing the valuation of a variable one by one (and replacing the variable with its valuation). Starting from the solution pool, the concretization process will always terminate with a concrete solution generated [9].

5 Tool Support and Case Study

SLSE has been completely implemented in a Java library called SUSHI [8]. SUSHI can be called as a back-end constraint solver by model checkers and symbolic execution engines. This section presents the experimental results and case study examples on its applications.



ID	Equation
eq1	$x \circ \mathtt{a}\{n,n\} \equiv (\mathtt{a}\|\mathtt{b})\{2n,2n\}$
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	$x^+_{\mathtt{a}^+\to\mathtt{b}\{n,n\}} \equiv \mathtt{b}\{2n,2n\}$
eq5	$\mathtt{uname='} \circ x^-_{\to\text{''}}[0,n] \circ \mathtt{'\ pwd='} \equiv \mathtt{uname='[\hat{}'|'']*'}$

Fig. 3. Running Statistics

5.1 Experiments

SUSHI relies on `dk.bricks.automaton` package [21] for manipulating FSA and a self-made package for supporting FST operations. In Figure 3 we listed five SLSE equations for stress-testing the SUSHI package and the running statistics. Note that each equation is parametrized by an integer n. For clarity, constant words are not wrapped with double quotation.

Clearly, the sample set covers `substring`, `replacement`, `concatenation`, and the hybrid use of these operations. In the experiment, the value of n ranges between 1 to 50. For example, when n is 50, the length of right hand side of `eq4` is 100. In Figure 3, the first diagram presents the max number of transitions of FSTs used in solving the equation. The second diagram presents the size of the max FSA used for representing solution pool(s). The last diagram shows the time spent on running each test.

Notice that for `eq1` and `eq2`, FST is not involved in the solution process, thus the max size of FST for these two equations is 0. Interestingly FST is used in solving `eq3` for the forward computing of $y[0, n]$. Another observation is that the complexity of the computation is mainly decided by the the complexity of the automata structure of the resulting solution. For example, the solution cost of `eq4` is higher than `eq5`. This is because when solving the atomic case of regular replacement, the cost of `eq4` is higher than `eq5`, as the RHS (of the replacement equation for `eq4`) is much more complex.

5.2 Case Study

We present two case study examples of SUSHI on discovering delicate XSS and SQL injection vulnerabilities.

Generating XSS Exploits: In many cases, when automated program analysis tools are not available (e.g., for handling rich and loosely typed languages such as JavaScript), SUSHI is a handy tool for manually analyzing programs. In the following we give one example of analyzing one recently discovered XSS vulnerability [17] in Adobe Flex SDK 3.3.

In Adobe Flex SDK, 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 [22] could identify 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.

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 [17]. Note that the first double quote is necessary for the exploit to work.

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

In the following, we briefly describe the SLSE equation constructed for generating the exploit. The SLSE equation is manually constructed, however, the construction can be automated if model checker or symbolic execution tools for JavaScript exist. In addition, the right hand side of the equation (a collection of attack patterns) can be easily reused.

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

The LHS of the equation is a conjunction of three strings, two constant words and one variable. The variable represents the unsanitized user input. The two constant words represents the other parts of the parameters collected and combined by the vulnerable Javascript code snippet. The size of LHS is 445 characters long. The RHS is a conjunction of the following attack patterns and filter rules as shown in the following.

The `attack` pattern is straightforward. It requires that the JavaScript alert() function eventually shows up in the combined output. Note that in the above table we omitted details of escaping forms, e.g., "`(`" should be actually escaped because it is a control symbol in java regex. Then the attack pattern has to be bounded by a number of rules for precisely generating the attack string: (1) the `EffectiveScript` rule forbids the JavaScript snippet to be embedded in any HTML attribute definition (thus ineffective); and, (2) the `MatchTag` rule requires that an HTML beginning tag must be matched by an ending tag (in our case the "`<embed>`" tag is the only one involved). Clearly, the above rules are general and can be applied to analyzing other XSS attacks. The cost of finding the shortest solution is shown below.

FST_States	FST_Trans	FSA_States	FSA_Trans	Time (seconds)
0	0	272	4217	74.109

Bypassing Password Checking: The second case study `BadLogin` is adapted from our previous work [6]. Notice that the proposed informal algorithm in the same paper cannot generate all possible attack strings, while our new theory introduced in this paper can. Consider the two program snippets in Listing 5.1.

The first snippet constructs a SQL query for authenticating a user by comparing the user supplied password with the information from the back-end database. It calls a `massage()` function (snippet 2) for sanitizing user input. Since single quote characters are used frequently by hackers, the `massage()` function replaces each single quote character with its escaping form (a sequence of two single quotes). Then it restricts the user input to a maximal length of 16.

```
//snippet 1.
"SELECT uname, pwd FROM users \n WHERE uname='"
+ massage(sUname) + "' AND pwd='" + massage(sPwd) + "'"

//snippet 2.
String massage(String strInput) {
 String sOut = strInput.replaceAll("'","''");
 if(sOut.length()>16) return sOut.substring(0,16);
 else return sOut;
}
```

Listing 5.1. Vulnerable Authentication

The length restriction protection, however, leads to vulnerability. Solving the following SLSE can directly lead to attack strings.

$$\texttt{uname='} \circ x\overline{\rightarrow}''[0,16] \circ \texttt{' AND pwd='} \circ y\overline{\rightarrow}''[0,16] \circ \texttt{'} \equiv \texttt{uname='([^']|'')*' OR *uname<>''}$$

The LHS is a concatenation of five string terms, with the $\circ$ operator denoting concatenation. The second term $x\overline{\rightarrow}''[0,16]$ represents the massaged user input (after replacement of single quote and substring operation). Similar is the fourth term $y\overline{\rightarrow}''[0,16]$. Note that each constant word is not delineated by double quotes for clarity. The RHS is a regular expression that represents an attack pattern. It asks: after applying the string massage operations on the user input (represented by variables x and y), is it feasible to bypass password checking and make the `WHERE` clause of the SQL query essentially a tautology in practice (by "`OR uname<>''`").

In [6], a pair of attack strings of length 16 are given. By using the algorithm presented in this paper, we are able to generate the *shortest* attack strings (not wrapped by double quotes) as shown below.

```
x = a''''''''
y = '' OR uname<>'
```

By applying the `massage()` function on x (i.e., `sUname`), each of the 8 single quotes in x 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 5.1). 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.

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

The delicate vulnerability discussed in this section cannot be discovered by black-box testing like [12, 5, 23], because without prior knowledge of the implementation of `massage()`, it is very hard to craft the malicious strings that could pass the sanity check performed by `massage()`. The cost of solving the equation is displayed as below. The tables displays the max size of FSAs/FSTs used in the solution process.

FST_States	FST_Trans	FSA_States	FSA_Trans	Time (seconds)
238	1634	17	62	1.39

6 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 [3], security vulnerability scanning [6, 10], and web application verification [25, 1]. 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. While most of the related work (e.g., [3, 4, 16, 25, 1]) fall into the category of forward analysis, our preliminary work [6, 7] and the research presented in this paper are backward.

It is worthy of note that, unlike symbolic model checking on Boolean programs (e.g., using BDD) and integer programs (e.g., using Presburger arithmetic), where backward analysis can be easily leveraged from forward analysis via the use of *existential quantification*, there is a huge gap between the *forward* and *backward* image computations for strings. Concerning forward analysis, the main focus is on fixpoint computation (or approximation). For example, Christensen *et al.* [4] used Mohri-Nederhof algorithm [20] to approximate from context-free languages to regular. Yu, Bultan, and Ibarra achieved forward fixpoint computation via widening technique for multi-tape automata [25]. The backward analysis of string equation systems can be very different as solving string constraints can be very challenging. Compared with forward string analysis, this research is able to generate attack strings as hard-evidence of a vulnerability. However, it suffers from false-negatives, i.e., there are cases that vulnerabilities are ignored by the analysis.

This paper adopts an automata based approach for solving simple linear string equations. SLSE can be regarded as a variation of the *word equation* problem [18]. Note that in a word equation, only word concatenation is allowed. In SLSE, various popular `java.regex` operations are supported. It is proved by Makanin that word problem is decidable and NP-hard [19]. However, extension of word equations can easily lead to undecidability. For example, the $\forall\exists^3$-theory of concatenation and word length predicates (according to [18]). Our intuition is that for many scenarios in web security, certain fragments of string equations suffice, e.g., the SLSE framework proposed in this paper, the fixed-length core string language in [1], and the unbounded string and size analysis [25].

Solving string constraints is one of the many directions for tackling command injection attacks (e.g., tainted analysis [22], forward string analysis [4], run-time hardening [11]), black-box testing [12]). Clearly, the string constraint solving technique can be applied to handling other problems in software engineering (e.g., symbolic model checking and symbolic execution).

SUSHI is a continuation of our earlier efforts of building a unified symbolic execution framework [6, 7] for detecting command injection attacks. There are similar efforts in the area. For example, Yu *et al.* proposed language based replacement in combining forward and backward string analysis [24]. Kieżrun *et al.* used symbolic tracking of taint information for discovering SQL injection

and XSS attacks [14]. Brumley *et al.* applied symbolic execution for analyzing binary program and can identify attack signatures for buffer overflow [2]. The uniqueness of SUSHI is its ability to precisely model various string substitution semantics in java `regex`, which avoids false positives during security analysis.

7 Conclusion

This paper introduces a general framework called *string equation* for modeling attack patterns. We show that a fragment called Simple Linear String Equation can be solved using automata based approach. Finite state transducer is used for precisely modeling several different semantics of regular substitution. The SLSE constraint solver is implemented and it is applied to analyzing security of Java web applications. The experimental results showed that the SLSE constraint solver works efficiently in practice and can discover security vulnerabilities that are hidden deeply in user input validation code.

References

1. Nikolaj Bjørner, Nikolai Tillmann, and Andrei 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. David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 311–325, 2007.
3. A. Simon 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.
4. Aske Simon Christensen, Anders Møller, and Michael 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/`.
5. CIRT INC. Nikto. *available at* `http://www.cirt.net/nikto2`.
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. X. Fu and K. Qian. SAFELI: SQL injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on testing, analysis, and verification of web services and applications (TAV-WEB 2008)*, pages 34–39, 2008.
8. Xiang Fu. Sushi - a solver for single linear string equations. `http://people.hofstra.edu/Xiang_Fu/XiangFu/projects.php`, 2009.
9. Xiang Fu and Chung chih Li. On regular replacement operators. `http://people.hofstra.edu/Xiang_Fu/XiangFu/publications/techrep09.pdf`, 2009.
10. 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.

11. 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.
12. 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.
13. Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistic*, 20(3):331–378, 1994.
14. Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09, Proceedings of the 30th International Conference on Software Engineering*, Vancouver, BC, Canada, May 20–22, 2009.
15. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
16. Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*, August 2006.
17. 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`, 2005.
18. M. Lothaire. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.
19. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sb.*, 32:129–198, 1977.
20. M. Mohri and M. J. Nederhof. Regular approximation of context-free grammars through transformation. *Robustness in Language and Speech Technology*, pages 153–163, 2001.
21. A. Møller. The dk.brics.automaton package. *available at* `http://www.brics.dk/automaton/`.
22. 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.
23. SPI Dynamics. Webinspect: Security throughout the application lifecycle. Datasheet. `http://www.spidynamics.com/assets/documents/WebInspect_DataSheets.pdf`.
24. Fang Yu, Muath Alkhalaf, and Tevfik 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.
25. Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proceedings of the 15th International Conference on Tools AND Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 322–336. Springer, 2009.