A Decision Procedure for Subset Constraints over Regular Languages *

Pieter Hooimeijer and Westley Weimer
University of Virginia
{pieter, weimer}@cs.virginia.edu

Abstract

Reasoning about string variables, in particular program inputs, is an important aspect of many program analyses and testing frameworks. Program inputs invariably arrive as strings, and are often manipulated using high-level string operations such as equality checks, regular expression matching, and string concatenation. It is difficult to reason about these operations because they are not well-integrated into current constraint solvers.

We present a decision procedure that solves systems of equations over regular language variables. Given such a system of constraints, our algorithm finds satisfying assignments for the variables in the system. We define this problem formally and render a mechanized correctness proof of the core of the algorithm. We evaluate its scalability and practical utility by applying it to the problem of automatically finding inputs that cause SQL injection vulnerabilities.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; F.3.1 [*Logics and Meanings*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Algorithms, Languages, Theory, Verification

1. Introduction

A large class of programs uses string variables to produce structured output, such as XML, SQL, or code [42]. Bugs in these programs can often be characterized by the subversion of that intended structure [41]. Two compelling examples of this type of bug are SQL injection and cross-site scripting vulnerabilities. These vulnerabilities are common; together they accounted for 35.5% of reported vulnerabilities in 2006 [21]. Reasoning about these bugs (e.g., for static detection) often involves the explicit modeling of possible runtime string values [32, 43, 44, 47].

* This research was supported in part by National Science Foundation Grants CNS 0627523 and CNS 0716478 and Air Force Office of Scientific Research grant BAA 06-028, as well as gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

PLDI'09, June 15–20, 2009, Dublin, Ireland.

We present a novel decision procedure for solving equations that involve sets of strings. We refer to the satisfaction problem for these equations as the *Regular Matching Assignments (RMA)* problem. Our decision procedure computes satisfying assignments for systems of equations that include regular language variables, concatenation, and language inclusion constraints. It can be used as a constraint solver to "push back" constraints across string operations. Many string variable operations, in particular equality checks and and regular expression matches, can be translated directly to an RMA instance.

The use of decision procedures is pervasive for many classes of program analysis and automatic testing; the common practice of querying an external decision procedure to reason about pointer aliases [34, 39] is a standard example. A decision procedure typically deals with a particular *theory*, such as linear arithmetic, uninterpreted functions, or bitwise operations. There are well-known frameworks for combining decisions procedures (e.g., [36]).

Decision procedures are often associated with *constraint solvers*. Formally, the constraint satisfaction problem returns a satisfying assignment while a decision problem returns a boolean; in practice many decision procedures return witnesses as well. The *satisfiability modulo theories* problem is a generalization of SAT [33, 48] that includes theories as well as literals and first-order logic [13, 40].

Recent work in this area has focused on adding theories, such as support for reasoning about bit vectors [13]. We similarly offer a procedure that allows for high-level reasoning about a theory of sets of strings. We foresee several immediate applications of our algorithm:

- **Directed randomized testing.** A number of recent projects use simultaneous concrete and symbolic execution to automate testing [16, 17, 18, 30, 38]. This is commonly referred to as *concolic testing* or *whitebox fuzzing*. These techniques rely heavily on constraint solvers to help "reverse engineer" inputs that lead to a given symbolic execution state and exercise a desired program path. Dealing with strings at a high level could significantly improve the precision of the constraint solving step in these frameworks.
- **Indicative testcase and path slice generation.** Bug reports, in particular those that are automatically generated, often go unaddressed for longer if the report does not include an indicative testcase [22, 46]. Program analyses that detect bugs can often be extended to generate indicative testcases [4]. Additionally, a program slice that elides irrelevant statements may further help a developer to understand a bug report [23].

In this paper, we show how our decision procedure can be used to extend the output of an existing bug finder [43] for SQL injection vulnerabilities. We use the output of that tool to generate actual

```
1  $newsid = $_POST['posted_newsid'];
2  if (!preg_match('/[\d]+$/', $newsid)) {
3    unp_msgBox('Invalid article news ID.');
4    exit;
5  }
6  $newsid = "nid_" . $newsid ;
7  $idnews = query("SELECT * FROM 'news'".
8                  "WHERE newsid='$newsid'");
```

Figure 1. SQL code injection vulnerability example adapted from Utopia News Pro. The `$_POST` mapping holds untrusted user-submitted data.

HTTP GET and POST parameters that would exploit the reported vulnerability.
The main contributions of this paper are:

- A formal decision procedure for regular language variables subject to concatenation and subset constraints.
- A mechanized proof of the correctness of the core of that decision procedure.
- A concrete example that demonstrates how our analysis can be used directly to address a program analysis problem: finding inputs for SQL code injection vulnerabilities. We include experimental evidence on 30,000 lines of code with 17 confirmed vulnerabilities to demonstrate the utility of our procedure.

The structure of this paper is as follows. In Section 2 we provide example code that exhibits an SQL injection vulnerability, and show how our decision procedure might be applied. Section 3 formally presents our decision procedure, building up from the Concatenation-Intersection problem (Section 3.2) to general systems of subset constraints (Section 3.4). Section 4 reports experimental results, and Section 5 briefly surveys closely-related work.

2. Motivating Example

In this section, we present a code fragment adapted from an existing application written in PHP. This code exhibits an SQL injection vulnerability, and demonstrates the need for analyses that are able to reason about the run-time values of string variables. More concretely, the code leads directly to a series of equations that our decision procedure can solve. The solutions are regular languages describing a set of inputs that exploit the SQL injection vulnerability; the inputs can be used to construct testcases and understand the defect. While our decision procedure is more widely applicable (e.g., to cross-site scripting or XML generation [21, 42]), we will use SQL injection as a running example.

Figure 1 shows a code fragment adapted from Utopia News Pro, a news management web service written in PHP. The `$_POST` array holds values that are submitted by the user as part of an HTTP request. A number of static analyses will (correctly) detect a potential vulnerability on line 7. The check on line 2 is designed to limit `$newsid` to numbers: `[\d]+` is a regular expression for a non-empty sequence of consecutive digits.

The `preg_match` function uses the delimiters `$` and `^` to match the end and the beginning of the string respectively. However, the check on line 2 is missing the `^` marker. Thus it is possible that the query sent on line 7 might be, for example, `"SELECT * from 'news' WHERE newsid='nid_' OR 1=1 ; DROP 'news' -- 9'"`. That particular query returns all entries in the `news` table to the attacker, and then deletes the table (the `--` begins a comment in SQL). Although the vulnerability is real, it may not be obvious to developers how an untrusted user can trigger it. For example, setting `posted_newsid` to `"' OR 1=1 ; DROP 'news' --"` fails to trigger it, instead causing the program to exit on line 4.

$$
\begin{array}{lcll}
S & ::= & E \subseteq C & \text{subset constraint} \\
E & ::= & E \circ E & \text{language concatenation} \\
 & | & C & \\
 & | & V & \\
 & & & \\
C & ::= & c_1 | \ldots | c_n & \text{constants} \\
V & ::= & v_1 | \ldots | v_m & \text{variables}
\end{array}
$$

Figure 2. Grammar for subset constraints over regular languages. The right-hand side (some element of C) is a single constant. The constants $c_1 \ldots c_n$ and variables $v_1 \ldots v_m$ each represent a regular language. The goal is to find satisfying assignments for the variables.

Conventional development relies heavily on regression testing and reproducible defect reports; a testcase demonstrating the vulnerability makes it more likely that the defect will be fixed [22, 46]. We therefore wish to form a testcase that exhibits the problem by generating values for input variables, such as:

```
posted_newsid = ' OR 1=1 ; DROP 'news' -- 9
posted_userid = a
```

To find this set of input values, we consider the constraints imposed by the code:

- The input set must pass the (incomplete) safety check on line 2.
- The input set must result in an exploit of interest on line 7.

This problem can be phrased as a constraint system over the string variables. In addition to the input set, our algorithm could reasonably be extended to produce a slice of the program with respect to the values that end up in the subverted query. In this case, the slice includes only lines 1 and 2, helping the developer locate potential causes of the error [4]. The particular actions taken by the generated exploit (e.g., whether all entries are returned or a table is dropped or modified) are a secondary concern. Instead, we want to allow analyses to detect the problem and generate a test case that includes string input values and a viable execution path through the program that triggers the vulnerability.

Finally, note that if the program in Figure 1 were fixed to use proper filtering, our algorithm would indicate that language of vulnerable strings for `posted_userid` is empty (i.e., that there is no bug). With this running example in mind, we will now formally define the problem of interest and give our algorithm for solving it.

3. Constraint Solving Over Regular Languages

In the following sections, we present our decision procedure for subset and concatenation constraints over regular languages. In Section 3.1, we provide a formal problem definition. The *Regular Matching Assignments (RMA)* problem defines the language constraints of interest, and also what constitutes a satisfying assignment for such a system of equations. Next, in Section 3.2, we define the *Concatenation-Intersection (CI)* problem, which we show to be a subclass of RMA. We provide an algorithm for solving instances of CI, and prove it correct in Section 3.3. In Section 3.4, finally, we extend the CI algorithm to general instances of RMA.

3.1 The Regular Matching Assignments Problem

In this section we define the *Regular Matching Assignments (RMA)* problem. The goal is to find satisfying assignments for certain systems of equations over regular languages. These systems consist of some number of constant languages and some number of language variables, combined using subset constraints and language concatenation.

More concretely, the example introduced in Section 2 can be expressed as the following set of constraints:

$$v_1 \subseteq c_1 \quad c_2 \circ v_1 \subseteq c_3$$

where c_1 corresponds to the input filtering on line 2, c_2 corresponds to the string constant `nid_` on line 6, and c_3 corresponds to undesired SQL queries. If we solve this system, then variable v_1 is the set of user inputs that demonstrate the vulnerability. If this set is empty, then the code is not vulnerable. If v_1 is nonempty, then we can use it to understand the problem and to generate testcases.

Figure 2 provides the general form of the *subset constraints* of interest. Formally, let an RMA problem instance $I = \{s_1, \ldots, s_p\}$ be a set of constraints over a shared set of variables $\{v_1, \ldots, v_m\}$. Each element $s_i \in I$ is a subset constraint of the form $e_i \subseteq c_i$, with e_i derivable from E in Figure 2 and c_i derivable from C. We write $A = [v_1 \leftarrow x_1, \ldots, v_m \leftarrow x_m]$ for an assignment of regular languages $\{x_1, \ldots, x_m\}$ to variables $\{v_1, \ldots, v_m\}$; let $A[v_i] = x_i$. Finally, let $L(r)$ denote the language of regular expression r.

If d is a variable-free expression derivable from E in Figure 2, then let $[\![d]\!]$ denote the regular language obtained by evaluating d. If d' is an expression with variables $\{v_1, \ldots, v_m\}$, then let $[\![d']\!]_A$ be the regular language obtained by first substituting all occurrences of $\{v_1, \ldots, v_m\}$ in d' with their corresponding assignments in A. We say that an assignment A *satisfies* I if and only if:

1. **Satisfying:** $1 \leq i \leq p \Rightarrow [\![e_i]\!]_A \subseteq [\![c_i]\!]$; and
2. **Maximal:** for each variable v_i, the corresponding language $A[v_i]$ cannot be extended without violating (1).

Given a problem instance I, the RMA problem requires either (1) a satisfying assignment; or (2) a message that no satisfying assignment exists. In some cases we may, additionally, be interested in all unique satisfying assignments; in Section 3.2 we show that the number of assignments is finite.

3.1.1 Examples

Consider the following example over the alphabet symbols x, y.

$$v_1 \subseteq L((\mathsf{xx})^+\mathsf{y}) \quad v_1 \subseteq L(\mathsf{x}^*\mathsf{y})$$

The correct satisfying assignment for this set of equations is $[v_1 \leftarrow L((\mathsf{xx})^+\mathsf{y})]$. The first condition, *Satisfying*, ensures that A assigns regular languages to language variables in a way that respects the input system of equations. In the example, the potential solution $A = [v_1 \leftarrow L(\mathsf{xy})]$ fails to satisfy the condition because $[\![v_1]\!]_A = L(\mathsf{xy}) \not\subseteq L((\mathsf{xx})^+\mathsf{y})$. The second condition, *Maximal*, prevents solutions that do not capture enough information. In the example, the potential solution $[v_1 \leftarrow \emptyset]$ satisfies condition one but not two; it is not maximal because it can be extended to, for example, $[v_1 \leftarrow L(\mathsf{xxy})]$.

RMA instances may have more than one unique satisfying assignment. For example, consider the following system:

$$v_1 \subseteq L(\mathsf{x}(\mathsf{yy})^+)$$
$$v_2 \subseteq L((\mathsf{yy})^*\mathsf{z})$$
$$v_1 \circ v_2 \subseteq L(\mathsf{xyyz}|\mathsf{xyyyyz})$$

This set of constraints has two *disjunctive* satisfying assignments:

$$A_1 = [v_1 \leftarrow L(\mathsf{xyy}), v_2 \leftarrow L(\mathsf{z}|\mathsf{yyz})]$$
$$A_2 = [v_1 \leftarrow L(\mathsf{x}(\mathsf{yy}|\mathsf{yyyy})), v_2 \leftarrow L(\mathsf{z})]$$

Both A_1 and A_2 satisfy the *satisfying* and *maximal* properties. They are also inherently disjunctive, however; it is not possible to "merge" A_1 and A_2 without violating one or both properties.

```
1: concat_intersect(c1, c2, c3) =
2:   Input: Machines M1, M2, M3 for c1, c2, c3;
3:          each machine Mj = ⟨Qj, Σ, δj, sj, fj⟩
4:   Output: Set of assignments; each Ai = [v1 ← xi', v2 ← xi'']
5:   // Construct intermediate automata
6:   let l4 = c1 ∘ c2 s.t. M4 = ⟨Q1 ∪ Q2, Σ, δ4, s1, f2⟩
7:   let l5 = l4 ∩ c3 s.t. M5 =
8:              ⟨(Q1 ∪ Q2) × Q3, Σ, δ5, s1 s3, f2 f3⟩
9:   // Enumerate solutions
10:  let Qlhs = {f1 q' | q' ∈ Q3} ⊆ Q5
11:  let Qrhs = {s2 q' | q' ∈ Q3} ⊆ Q5
12:  foreach (qa, qb) ∈ Qlhs × Qrhs s.t. qb ∈ δ5(qa, ε) do
13:    let M1' = induce_from_final(M5, q1)
14:    let M2' = induce_from_start(M5, q2)
15:    output [v1 ← M1', v2 ← M2']
16:  end for
```

Figure 3. Constraint solving for intersection across concatenation. The algorithm relies on basic operations over NFAs: concatenation using a single ϵ-transition (line 6) and the cross-product construction for intersection (line 7–8). The two induce functions are described in the text.

3.1.2 Extensions

Our definition of RMA (and the algorithm we provide to solve it) can be readily extended to support additional operations, such as union or substring indexing. For example, substring indexing might be used to restrict the language of a variable to strings of a specified length n (to model length checks in code). This could be implemented using basic operations on nondeterministic finite state automata that are similar to the ones already implemented. Many other features, however, would make the RMA problem undecidable in general (e.g., [27]). We instead focus on a decidable theory with a provably correct core algorithm, and leave additional features for future work. In Section 4 we show that, without additional features, our decision procedure can be used to solve a real-world problem.

3.2 The Concatenation-Intersection Problem

Rather than solving the general RMA problem directly, we will first consider a restricted form involving only the concatenation of two variables that each have a subset constraint. In the next subsections we will present this restricted problem, and in Section 3.4 we will use our solution to it to build a full solution for the RMA problem.

We define the *Concatenation-Intersection (CI)* problem as a subcase of the RMA problem, of the following form:

$$v_1 \subseteq c_1 \quad v_2 \subseteq c_2 \quad v_1 \circ v_2 \subseteq c_3$$

Because the form of these constraints is fixed, we define a CI problem instance strictly in terms of the constants c_1, c_2, and c_3. Given three regular languages c_1, c_2, and c_3, the CI problem requires the set $S = \{A_1, \ldots, A_n\}$ of satisfying assignments, where each A_i is of the form $[v_1 \leftarrow x_i', v_2 \leftarrow x_i'']$. More explicitly, we require that S satisfy the following properties:

1. **Regular:** $1 \leq i \leq n \Rightarrow A_i[v_1]$ and $A_i[v_2]$ are regular.
2. **Satisfying:** This corresponds directly to the *Satisfying* condition for RMA:

$$1 \leq i \leq n \Rightarrow [\![v_1]\!]_{A_i} \subseteq [\![c_1]\!] \wedge [\![v_2]\!]_{A_i} \subseteq [\![c_2]\!] \wedge [\![v_1 \circ v_2]\!]_{A_i} \subseteq [\![c_3]\!]$$

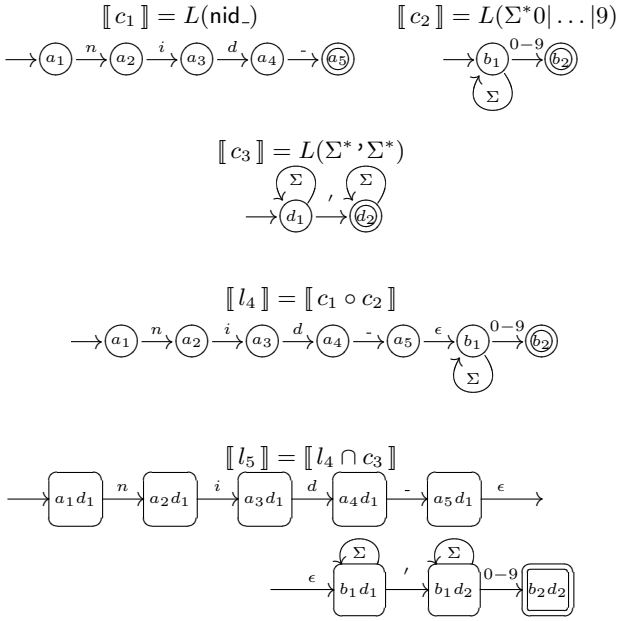

Figure 4. The intermediate finite state automata for the concat_intersect algorithm when applied to the motivating example. c_1 represents the string constant `nid_`, c_2 represents the (incorrect) input filtering on line 2, and c_3 is the language of strings that contain a single quote.

3. **All Solutions:** In addition, we want S to be nontrivial, since the *Satisfying* condition can be trivially satisfied by $S = \emptyset$:

$$\forall w \in [\![(c_1 \circ c_2) \cap c_3]\!], \exists 1 \leq i \leq n \text{ s.t. } w \in [\![v_1 \circ v_2]\!]_{A_i}$$

Figure 3 provides high-level pseudocode for finding S. The algorithm uses operations on nondeterministic finite state automata, and we write M_i for the machine corresponding to each input language c_i or intermediate language l_i. To find S, we use the structure of the NFA M_5 (line 7–8) that recognizes $[\![l_5]\!] = [\![(c_1 \circ c_2) \cap c_3]\!]$. Without loss of generality, we assume that each NFA M_i has a single start state $s_i \in Q_i$ and a single final state $f_i \in Q_i$. Note that we do not assume implicit ϵ-transitions from each state to itself.

Figure 4 shows the CI algorithm applied to the running example introduced in Section 2. The input languages c_1 and c_2 correspond to the concatenation `$newsid = "nid_" . $newsid` on line 6 of Figure 1, while the input c_3 corresponds to the set of strings that contain at least one quote — which is one common approximation for an unsafe SQL query [43, 44].

The machines for l_4 and l_5 in Figure 4 correspond to the machines constructed on lines 6 and 7–8 of Figure 3. The algorithm first constructs a machine for $[\![l_4]\!] = [\![c_1 \circ c_2]\!]$ using a single ϵ-transition between f_1 and s_2. Next, we use the cross-product construction to create the machine that corresponds to $[\![l_5]\!] = [\![l_4 \cap c_3]\!]$. The set of states Q_5 for this machine corresponds to tuples in the set $(Q_1 \cup Q_2) \times Q_3$; we write $q_x q_y \in Q_5$ for the state that corresponds to $q_x \in (Q_1 \cup Q_2)$ and $q_y \in Q_3$. The transition function δ_5 is defined in the usual way.

Having constructed M_5, we use the structure of the machine to find NFAs that represent the satisfying assignments. Intuitively, we slice up the bigger machine M_5, which represents all solutions, into pairs of smaller machines, each of which represents a single satisfying assignment.

We are interested in those states $q_a q_b \in Q_5$ where q_a corresponds to the final state of M_1 (i.e., Q_{lhs} on line 10) or to the start state of M_2 (i.e., Q_{rhs} on line 11). Because of the way M_5 is constructed, any transitions from Q_{lhs} to Q_{rhs} must be ϵ-transitions that correspond to the original concatenation step on line 6 of Figure 3. For Figure 4, we have $Q_{\text{lhs}} = \{a_5 d_1\}$ and $Q_{\text{rhs}} = \{b_1 d_1\}$. We process each such ϵ-transition as follows:

- induce_from_final(M_5, q_1) (line 10) returns a copy of M_5 with q_1 marked as the only final state.
- induce_from_start(M_5, q_2) (line 11) returns a copy of M_5 with q_2 marked as the only start state.

We output each such solution pair. Note that, on line 15, if either M_1' or M_2' describe the empty language, then we reject that assignment.

The machine for l_5 in Figure 4 has exactly one ϵ-transition of interest. Consequently, the solution set consists of a assignment $A_1 = [v_1 \leftarrow x_1', v_2 \leftarrow x_1'']$. x_1' corresponds to the machine for l_5 with state $a_5 d_1$ set as the only final state. $[\![x_1']\!] = L(\text{nid\_})$, as desired. The more interesting result is x_1'', which is the machine with start state $b_1 d_1$ and final state $b_2 d_2$. The language of x_1'' captures exactly the strings that exploit the faulty safety check on line 2 of Figure 1: all strings that contain a single quote and end with a digit.

3.3 Correctness of the Concat-Intersect Algorithm

Having presented our high-level algorithm for solving the CI problem, we now sketch the structure of our proof of its correctness. We have formally modeled strings, state machines, our algorithm, and the desired correctness properties in version 8.1 of the Coq formal proof management system [6, 12]; proofs in Coq can be mechanically verified.

At the low level, our proofs proceed by induction on the length of the strings that satisfy specific properties; we provide more detail for each individual correctness property. We say q reaches q' on s if there is a path from state q to state q' that consumes the string s.

Formally: for all regular languages c_1, c_2, and c_3, if $S = \text{CI}(c_1, c_2, c_3)$ then for all elements $A_i \in S$, the following three conditions hold:

1. **Regular:** $A_i[v_1]$ and $A_i[v_2]$ are regular.
   This is a type preservation property: because the operations induce_from_final and induce_from_start return NFAs, the corresponding languages are by definition regular.
2. **Satisfying:** We prove this by showing that $\forall q \in Q_{\text{lhs}}, w \in \Sigma^*, s_5$ reaches q on string $w \Rightarrow w \in c_1$, and $\forall q' \in Q_{\text{rhs}}, w \in \Sigma^*$, q' reaches f_5 on $w \Rightarrow w \in c_2$.
3. **All Solutions:** We proceed by simultaneous induction on the structure of the machines M_3 and M_4; we show that $\forall w \in A_i[v_1]$, $w' \in A_i[v_2]$, s_5 reaches f_5 on $s \circ s'$ in machine M_5 (by traversing the epsilon transition selected for A_i). Note that, since the number of ϵ-transitions in M_5 is finite, the number of disjunctive solutions must also be finite.

Our proof is available on-line[1]; we believe that the presence of a mechanically checkable proof of correctness makes our algorithm attractive for use as a decision procedure or as part of a sound program analysis.

3.4 Solving General Systems of Subset Constraints

We now return to the problem of finding satisfying assignments for general regular language equations. At a high level, this requires

[1] `http://www.cs.virginia.edu/~ph4u/dprle/proof.php`

$$\frac{n = \mathsf{node}(c_i)}{\vdash c_i : n, \varnothing}\ E \to C \qquad \frac{n = \mathsf{node}(v_i)}{\vdash v_i : n, \varnothing}\ E \to V$$

$$\frac{\begin{array}{c} t \text{ is fresh} \\ \vdash e_0 : n_0, G_0 \\ \vdash e_1 : n_1, G_1 \\ G' = \{\mathsf{ConcatEdgePair}(n_0, n_1, t)\} \end{array}}{\vdash e_0 \circ e_1 \ : \ t, G_0 \cup G_1 \cup G'}\ E \to E \circ E$$

$$\frac{\vdash e \ : \ n, G}{\vdash e \subseteq c \ : \ n, G \cup \{\mathsf{SubsetEdge}(\mathsf{node}(c), n)\}}\ S \to E \subseteq C$$

Figure 5. Dependency graph generation rules. We process a regular language constraint by recursive descent of its derivation; each rule corresponds to a grammar production. The node function returns a vertex for each unique variable or constant. For systems of multiple constraints, we take the union the dependency graphs.

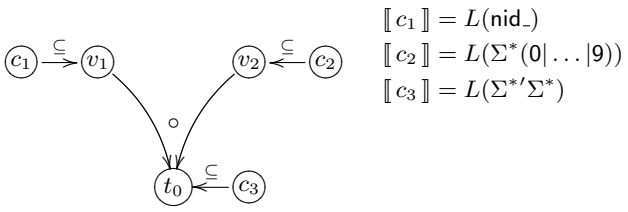

Figure 6. Example dependency graph. This graph corresponds to an instance of the Concatenation-Intersection problem defined in Section 3.2, using the assignments for the running SQL injection example from Section 2.

that we generalize the concat_intersect algorithm from Figure 3. We proceed as follows:

1. To impose an ordering on the operations of our algorithm, we create a *dependency graph* based on the structure of the given equations. We describe the graph generation process in Section 3.4.1.
2. Section 3.4.2 provides a worklist algorithm that applies the concat_intersect procedure inductively, while accounting for repeated variable instances. That is, if a variable occurs multiple times, a solution generated for it must be consistent with all of its uses.

3.4.1 Dependency Graph Generation

Our approach to constraint solving is conceptually related to the equality DAG approach found in cooperating decision procedures [36]. Each unique language variable or constant is associated with a node in the DAG. We construct the edges of the DAG by recursively processing the regular language equation. We present the processing rules here; the algorithm in Section 3.4.2 assumes a dependency graph as its input.

Given a regular language equation, we build the dependency graph by recursive descent of the derivation under the grammar of Figure 2. Figure 5 describes the generation rules as a collecting semantics. The rules are of the form:

$$\vdash e : n, G$$

where e is the right-hand side of a derivation step, n is the current dependency graph vertex, and G is the dependency graph. The node function returns a distinct vertex for each unique variable and constant. Each vertex represents a regular language; we write $[\![n]\!]$ for the language associated with vertex n. We model the graph G as a set of directed edges, of which there are two types:

- $\mathsf{SubsetEdge}(n_0, n_1)$ requires that $[\![n_1]\!] \subseteq [\![n_0]\!]$. In such an edge n_0 is a constant and n_1 is a language variable. We write such edges as $n_0 \to^{\subseteq} n_1$ and refer to them as $\subseteq$-edges.
- $\mathsf{ConcatEdgePair}(n_a, n_b, n_0)$ constrains the language $[\![n_0]\!]$ to strings in $[\![n_a]\!] \circ [\![n_b]\!]$. Each constraint has two edges $n_a \to^{\circ}_{l} n_0$ and $n_b \to^{\circ}_{r} n_0$ referred to as a $\circ$-edge pair.

The base case rules in Figure 5 are those for $E \to C$ and $E \to V$. All other rules extend the dependency graph through union. Note that the rule for $E \to E \circ E$ uses a fresh vertex t to represent the intermediate result of the concatenation. Finally, the top-level rule (for $S \to E \subseteq C$) adds a single $\subseteq$-edge from the right-hand side of the constraint c to the left-hand side.

Figure 6 shows the dependency graph for the example of Section 2. This graph corresponds an instance of the CI problem defined in Section 3.2, of the form:

$$v_1 \subseteq c_1 \quad v_2 \subseteq c_2 \quad v_1 \circ v_2 \subseteq c_3$$

Note that the edges in Figure 6 are strictly a description of the constraint system; they are *not* meant to indicate a strict dependence ordering. For example, changing $[\![c_3]\!]$ to be be $L(\mathsf{nid\_}'5)$ would require $[\![v_2]\!]$ to contain only the string $'5$, even through there is no forward path through the graph from c_3 to v_2.

3.4.2 Solving General Graphs

We now provide a general algorithm for the RMA problem defined in Section 3.1. Given a system of regular language equations, our algorithm returns the full set of disjunctive satisfying assignments from language variables to regular languages. The essence of our algorithm is the repeated application of a generalized (i.e., able to handle groups of nodes connected by concat edges) version of the concat_intersect algorithm.

The algorithm keeps a worklist of partially-processed dependency graphs along with mappings from vertices to finite state automata; the use of a worklist is necessary to handle disjunctive solutions. The initial worklist consists of the dependency graph that represents the regular language equation in full. The the initial node-to-NFA mapping returns Σ^* for vertices that represent a variable, and $[\![c_i]\!]$ for each constant c_i.

Figure 7 provides pseudocode for the algorithm, which is recursive. The algorithm consists of several high-level stages, which are applied iteratively:

1. On lines 3–8, we solve basic constraints. Many constraints can be resolved by eliminating vertices that represent constants in topological order. For example, the system

   $$v_1 \subseteq c_1 \quad v_1 \subseteq c_2 \quad v_2 \subseteq c_1 \quad v_2 \subseteq c_2$$

   can be processed by simply setting $A[v_2] = A[v_1] = [\![c_1 \cap c_2]\!]$. This step does not require any calls to the concat_intersect procedure described in Section 3.2. Also note that this step never generates more than one set of solutions.

   The sort_acyclic_nodes invocation on line 3 finds vertices that qualify for this treatment, and sorts them topologically. The reduce function performs NFA intersections (to satisfy subset constraints) and concatenations (to satisfy concatenation constraints), and removes nodes from the graph that have no further inbound constraints.
2. On lines 9–15, we apply the CI algorithm inductively to handle nodes of the graph that have both concatenation and subset

```
1: solve_dependency_graph(queue Q, node set S) =
2:   let ⟨G, F⟩ : graph × (node → NFA) = take from Q
3:   let N : node list = sort_acyclic_nodes(G)
4:   for 0 ≤ i < length(N) do
5:     let n : node = N[i]
6:     let ⟨G', F'⟩ : graph × (node → NFA) = reduce(n, G, F)
7:     F ← F'; G ← G'
8:   end for
9:   let C : node set = find_free_group(G)
10:  if |C| > 0 then
11:    let ⟨G', R⟩ : graph × (node → NFA) list = gci(C, G, F)
12:    G ← G'; F ← head(R)
13:    foreach r ∈ tail (R) do
14:      add ⟨G, r⟩ to end of Q
15:  end if
16:  if ∀s ∈ S. F[s] ≠ ∅ ∧ |G| = 0 then
17:    return F
18:  else if ∀s ∈ S. F[s] ≠ ∅ ∧ |G| > 0 then
19:    return solve_dependency_graph(⟨q, F⟩ :: Q, S)
20:  else if ∃s ∈ S s.t. F[s] = ∅ ∧ |Q| > 0 then
21:    return solve_dependency_graph(Q, S)
22:  else
23:    return no assignments found
```

Figure 7. Constraint solving algorithm for general dependency graphs over string variables. The algorithms uses a worklist of dependency graphs and node-to-NFA mappings. The graph represents the work that remains; successful termination occurs if all nodes are eliminated.

constraints. We refer to connected groups of those nodes as *CI-groups*; we define such groups formally in Section 3.4.3.

Each call to gci (for *generalized concat-intersect*) on line 11 eliminates a single CI-group from the dependency graph, yielding separate node-to-NFA mappings for each disjunctive solution. These solutions are added to the worklist on lines 13–14.

3. Lines 16–23 determine what to do next. We either (1) terminate successfully (line 17); (2) continue to solve the current graph (line 19); (3) attempt to solve a new graph from the worklist (line 21); or (4) terminate without having found a satisfying set of inputs (line 23).

3.4.3 Solving CI-Groups Integrally

We define a *CI-group* as any set of nodes in which every node in the set is connected by a $\circ$-edge to another node in the set. The directions of the $\circ$-edges do not matter. In Figure 9 the nodes $\{v_a, v_b, v_c, t_1, t_2\}$ form a CI-group; we will use this example to illustrate how these groups can be solved.

The purpose of the generalized concat-intersect (gci) procedure referenced in Figure 7 is to find a solution (i.e., a mapping from variables to NFAs) for the nodes involved in a CI-group. Since an RMA problem may admit multiple disjunctive solutions (see Section 3.1.1), the output of gci is a set of such solutions.

The gci algorithm solves a CI-group by repeatedly processing subset constraints and concatenation constraints. In the concat_intersect algorithm (Figure 3), the final solutions M_1' and M_2' were both sub-NFAs of a larger NFA. Similarly, the solution for one variable in a CI-group may be a sub-NFA of the solution for another variable. A variable may appear as the operand in more than one concatenation; in that case we must take care to find an assignment that satisfies all constraints simultaneously. The correctness of the gci algorithm is based on two key invariants: operation ordering and shared solution representation.

```
1: gci(node set C, graph G, node → NFA F) =
2:   let ordered : node list = topo_sort(C)
3:   let solution : node → ((node × subNFA) set) = empty
4:   foreach n : node ∈ ordered do
5:     handle_inbound_subset_constraints(n, G, F)
6:     if n has an outbound concat constraint to node m then
7:       handle_concat_constraint(n, m, G, F)
8:     foreach n' ∈ C do
9:       foreach (n'', states) ∈ solution[n'] do
10:        if n'' = n then
11:          update_tracking(solution[n'], n)
12:  end for
13:  let S : state pair set = emtpy
14:  foreach m ∈ C s.t. solution[m] is empty do
15:    S ← S ∪ all_combinations(m, F[m])
16:  end for
17:  return generate_NFA_mappings(solution, G, C, S)
```

Figure 8. Generalized concat-intersect algorithm for finding a set of disjunctive solutions for a group of nodes connected by $\circ$-edges. Inbound subset constraints are processed before concatenation constraints. The solution for a node n may be a sub-NFA S of another node m's solution; the solution mapping tracks this (i.e., $(m, S) \in \text{solution}[n]$).

The first invariant, *operation ordering*, requires that inbound subset constraints be handled before concatenation constraints. The importance of this ordering can be seen in Figure 6. Initially, $[\![v_1]\!] = [\![v_2]\!] = L(\Sigma^*)$. If we mistakenly process the concat edge first, we obtain $[\![t_0]\!] = L(\Sigma^*\Sigma^*)$. If we then process the subset edges, we obtain $[\![v_2]\!] = [\![c_2]\!]$, which is not correct — the correct solution, as described in Section 3.2, is $[\![v_2]\!] = L(\Sigma^{*\prime}\Sigma^*(0| \ldots |9))$. To obtain the correct solution and "push back" subset constraints through concatenations, we must process subset constraints first.

The second invariant, *shared solution representation*, ensures that updates to the NFA $F[v]$ representing the solution for a variable v are also reflected in updates to the solutions to all variables v' that are sub-NFAs of $F[v]$. In Figure 6, an update to the NFA for t_0 must also be reflected in the NFAs for v_1 and v_2. Our gci implementation maintains a shared pointer-based representation so that if the NFA for t_0 is changed (e.g., is subjected to the product construction to handle a subset constraint), then the NFAs for v_1 and v_2 are automatically updated.

These two invariants allow us to handle nested concatenation operations naturally. Consider the following system:

$$(v_1 \circ v_2) \circ v_3 \subseteq c_4 \quad v_1 \subseteq c_1$$
$$v_2 \subseteq c_2 \quad v_3 \subseteq c_3$$

with the parentheses added for clarity. In this case, the dependency graph will be several concatenations "tall." The final subset with $[\![c_4]\!]$, notably, can affect any of the variables v_1, v_2, and v_3. If the constraints are processed in the right order, the NFAs for v_1, v_2 and v_3 will all be represented as sub-NFAs of a single larger NFA.

Figure 8 provides high-level pseudocode for solving a single CI-group. The expected output is a set of node-to-NFA mappings (one mapping for each disjunctive solution). The algorithm works as follows:

1. The nodes are processed in topological order (line 2).
2. The solution representations for each node n are tracked in solution$[n]$. There will be multiple solution entries for a given node if that node is the operand for more than one concatenation. The set solution$[n]$ contains zero or more pairs of the form

(m, S), where m is another node and S is a sub-NFA selecting part or all of the NFA for the node m. Each such pair indicates that $[\![n]\!]$ should be constrained by the NFA for node $m \neq n$, limited to the states specified by S.

3. Each node n starts without any constraints (solution$[n]$ is empty, line 3). If $\exists(m, S) \in$ solution$[n]$ then the solution for node n is influenced by changes to the solution for node m and we say that m *influences* n.
4. If a node m has an inbound subset constraint $c \rightarrow^{\subseteq} m$, then any nodes influenced by m have their entry in solution updated to reflect the new machine for $[\![m]\!] \cap [\![c]\!]$. This is handle_inbound_subset_constraints on line 5 and the updates on lines 8–11.
5. If a node m is concatenated into another node t (line 6), then any nodes influenced by m will have their entry in solution updated to map to a sub-NFA of t (this maintains the shared solution representation invariant). The mapping for solution$[m]$ is extended to include the pair (t, S), so that m itself is now marked as influenced by t (with the appropriate subset of states S). This is handle_concat_constraint on line 7 and the updates on lines 8–11.
6. After all nodes have been processed (after line 12), there will be some number of non-influenced nodes that do not have any outbound $\circ$-edge pairs and are thus not in solution. For each such node n, the final solution is simply $F[n]$. Intuitively, these are the largest NFAs in the CI-group, since each concatenation and intersection increases the size of the resulting NFA.
7. The solution for each influenced node m (line 14) will refer to one or more non-influenced nodes, each reference coupled with an appropriate subset of states to take from the larger machine. Because of the structure for the grammar in Figure 2, there is always one non-influenced node; we use the NFAs of that node to generate the disjunctive solutions.

   Recall that, in the concat_intersect procedure of Figure 3, we generated disjunctive solutions by selecting ϵ-transitions from the machine M_5. To generalize this, we must generate a disjunctive solution for each *combination* of such ϵ-transitions in the non-influenced nodes' NFAs. The all_combinations invocation (Figure 8, line 15) generates these combinations, and the call to generate_NFA_mappings (line 17) generates a new node-to-NFA mapping for each such combination.

3.4.4 Example Execution

Figure 10 shows the intermediate automata generated by the gci procedure when applied to the dependency graph of Figure 9. The dashed lines in the NFAs for t_1 and t_2 mark the epsilon transitions for the $v_a \circ v_b$ and $v_b \circ v_c$ concatenations, respectively. After processing each of the nodes (i.e., on line 12 of Figure 8), the solution mapping is as follows:

$$
\begin{aligned}
v_a &\mapsto \{(t_1, \{a_1d_1, a_2d_2, \ldots\})\ \} \\
v_b &\mapsto \{(t_1, \{b_1d_4, b_1d_6, \ldots\});\ \ (t_2, \{b_1g_1, b_2g_2, \ldots\})\ \} \\
v_c &\mapsto \{(t_2, \{c_1g_3, c_1g_4, \ldots\})\ \} \\
t_1 &\mapsto \emptyset \\
t_2 &\mapsto \emptyset
\end{aligned}
$$

Note that that the machines for t_1 and t_2 each have two ϵ-transitions: t_1 connects the submachines for v_a to that of v_b, while t_2 connects submachines for v_b on the left-hand side to the submachines for v_c an the right-hand side. This yields a total of 2×2 candidate solutions. Further, the solution mapping shows us that v_b participates in both concatenations, so for each candidate solution we must ensure that $[\![v_b]\!]$ satisfies both constraints. This leaves two satisfying assignments:

1. $A_1 = [v_a \leftarrow L(\mathsf{op}^2), v_b \leftarrow L(\mathsf{p}^3\mathsf{q}^2), v_c \leftarrow L(\mathsf{q}^2\mathsf{r})]$
   This solution used $F[t_1] \equiv L(\mathsf{op}^5\mathsf{q}^2)$ and $F[t_2] \equiv L(\mathsf{p}^3\mathsf{q}^4\mathsf{r})$
2. $A_2 = [v_a \leftarrow L(\mathsf{op}^4), v_b \leftarrow L(\mathsf{pq}^2), v_c \leftarrow L(\mathsf{q}^2\mathsf{r})]$
   This solution used $F[t_1] \equiv L(\mathsf{op}^5\mathsf{q}^2)$ and $F[t_2] \equiv L(\mathsf{pq}^4\mathsf{r})$

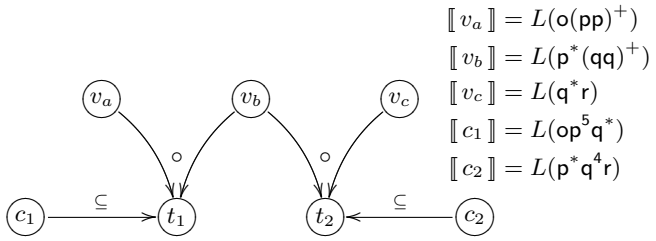

Figure 9. A partially-processed dependency graph that exhibits a CI-group (defined in the text). In this case, v_b is affected by both $c_1 \rightarrow^{\subseteq} t_1$ and $c_2 \rightarrow^{\subseteq} t_2$, making the two concatenations mutually dependent. The correct solution set for this graph includes all possible assignments to v_a and v_c for which there exists an assignment to v_b that simultaneously satisfies the constraints on t_1 and t_2.

3.5 Runtime Complexity

In this section we have defined the Regular Matching Assignments problem and presented a decision procedure for it. We now turn to the runtime complexity of the decision procedure. As before, we will first discuss the concat_intersect procedure presented in Section 3.2, and then generalize to the full algorithm of Section 3.4.2. It should be noted that the precise complexity of combined NFA operations is the subject of ongoing research [37]. We use worst-case NFA state space complexity (i.e., the number of NFA states visited during an operation) to represent the runtime complexity of our algorithm. This is a natural representation for low-level NFA operations that require visiting all states in one or more operands, such as the cross-product construction.

We express our analysis in terms of two variables: the number of disjoint solutions for a given system, and the total number of NFA states visited. In each case, we compute a worst-case upper bound. We refer to the size $|M_i|$ of an NFA M_i as the size of its state space; let Q be an upper bound on the size of any input NFA.

The concat_intersect algorithm performs one concatenation operation (line 6) and then computes machine M_5 using the cross-product construction (lines 7–8). The size of the concat machine is $|M_1| + |M_2| = O(Q)$, so constructing the intersection requires visiting $|M_3|(|M_1| + |M_2|) = O(Q^2)$ states. The number of solutions in the intersection language is bounded by $|M_3|$. This is because of the way Q_{lhs} and Q_{rhs} are defined in Figure 3; the proof sketch in Section 3.3 provides more detail.

This means that the total cost of enumerating all solutions eagerly, in terms of NFA states visited, is $|M_3| \times (|M_3|(|M_1| + |M_2|)) = O(Q^3)$. We note that, in practice, we can generate the first solution without having to enumerate the others; this is why we reason separately about machine size and number of possible solutions.

The argument thus far applies to dependency graphs of the form illustrated in Figure 6; we now extend it to general dependency graphs. We consider two cases: (1) one or both the $\circ$-operands c_1 and c_2 are the result of a previous call to the concat_intersect

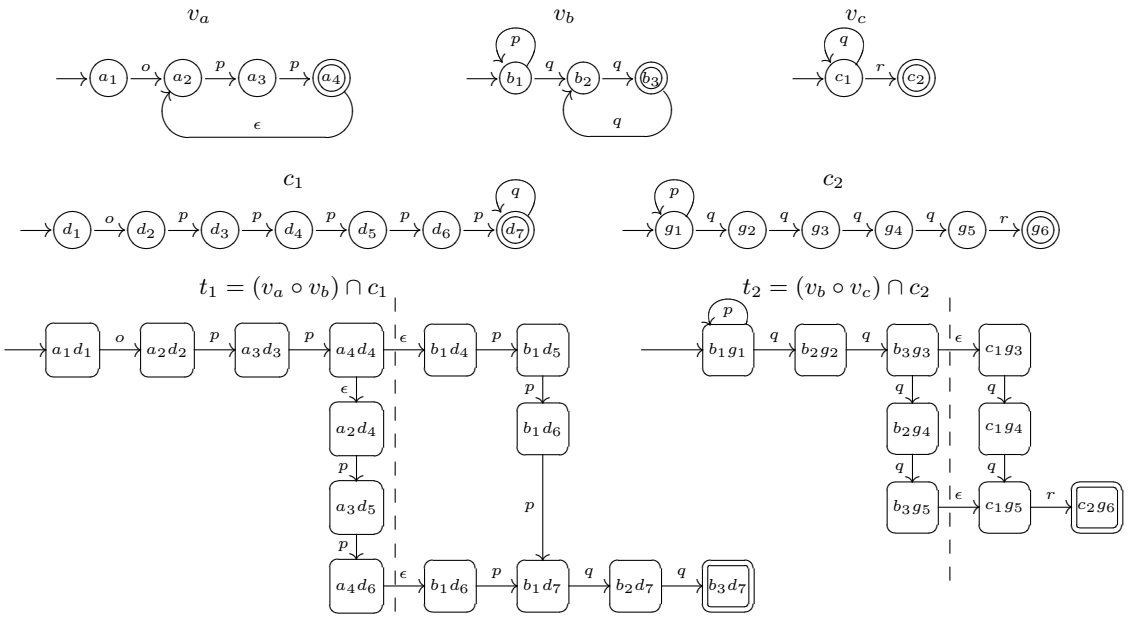

Figure 10. Intermediate automata for solving Figure 9. The gci procedure (Figure 8) finds disjunctive solutions that satisfy the constraints on t_1 and t_2 separately. It then considers all combinations of these solutions and outputs the solution combinations that have matching machines v_b.

procedure; and (2) the concatenation result t_0 is subject to more than one subset constraint.

In the first case, suppose v_1 is the result of a separate concat-intersect step; its NFA has size $O(Q^2)$ and there are $O(Q)$ possible assignments to v_1. The machine for $v_1 \circ v_2$ then has size $O(Q^2) + O(Q) = O(Q^2)$, yielding a total enumeration size of

$$\underbrace{O(Q^2)}_{\text{solutions}} \times \underbrace{O(Q^3)}_{\text{machine size}} = O(Q^5)$$

In the second case, we consider adding an additional subset constraint to the concatenation node t_0 in the graph of Figure 6. Note that $|M_3|$ occurs both as a factor in the number of states visited and the number of potential solutions. We add one additional subset constraint to the concatenation (i.e., $v_1 \circ v_2 \subseteq c_4$); we assume one additional edge $c_4 \rightarrow^{\subseteq} t_0$. The enumeration then requires visiting

$$\underbrace{|M_3||M_4|}_{\text{solutions}} \times \underbrace{(|M_3||M_4|(|M_1| + |M_2|))}_{\text{machine size}} = O(Q^5)$$

states.

Informally, we note that a single concat_intersect call requires visiting at most Q^3 NFA states. The total cost of solving a general constraint graph grows exponentially with the number of inductive calls to that procedure. For example, a system:

$$\begin{array}{rr} v_1 \subseteq c_1 & v_1 \circ v_2 \quad \subseteq c_4 \\ v_2 \subseteq c_2 & v_1 \circ v_2 \circ v_3 \quad \subseteq c_5 \\ v_3 \subseteq c_3 & \end{array}$$

requires two calls to concat_intersect. To enumerate the first solution for the whole system we must visit a total of $O(Q^3)$ NFA states; enumerating all possible solutions requires visiting $O(Q^5)$ states. In Section 4, we show that the algorithm, in spite of its exponential worst-case complexity, is efficient enough to be practical.

3.6 Regular-Language Constraint Solving Summary

In this section we have defined the general Regular Matching Assignments problem for equations of regular language variables and presented a decision procedure for it. We show how to construct a dependency graph and process parts of it in sequence to generate potentially-disjunctive solutions. The heart of our algorithm is the inductive application of our solution to the Concatenation-Intersection problem, which "pushes back" intersection constraint information through language concatenation. Having presented our algorithm and proved the correctness of its core, we now turn to an empirical evaluation of its efficiency and utility.

4. Experimental Results

Recently, much attention has been devoted to static techniques that detect and report potential SQL injection vulnerabilities (e.g., [24, 31, 47]). Attacks remain prevalent [8], however, and we believe that extending static analyses to include automatically generated test inputs would make it easier for programmers to address vulnerabilities. Without testcases, defect reports often go unaddressed for longer periods of time [22, 46], and time is particularly relevant for security vulnerabilities.

To test the practical utility and scalability of our decision procedure, we implemented a prototype that automatically generates violating inputs for given SQL injection vulnerabilities. These vulnerabilities allow undesirable user-supplied commands to be passed to the back-end database of a web application. Such attacks are quite common in practice: in 2006, SQL injection vulnerabilities made up 14% of reported vulnerabilities and were thus the second most commonly-reported security threat [21].

We extend an existing analysis by Wassermann and Su [43], which detects SQL injection vulnerabilities but does *not* automatically generate testcases. Since our decision procedure works on systems of regular language equations, we constructed a simple prototype program analysis that uses symbolic execution to set up a system of string variable constraints based on paths that lead to the defect reported by Wassermann and Su. We then apply our algorithm to solve for any variables that were part of an HTTP GET or POST request.

Name	Version	Files	LOC	Vulnerable
eve	1.0	8	905	1
utopia	1.3.0	24	5,438	4
warp	1.2.1	44	24,365	12

Figure 11. Programs in the Wassermann and Su [43] data set with more than one direct defect. The *vulnerable* column lists the number of files for which we generated user inputs leading to a potential vulnerability detected by the Wassermann and Su analysis; in our experiments we attempt to find inputs for the first vulnerability in each such file.

We ran our experiments on seventeen defect reports from three programs used by Wassermann and Su [43]. The programs are large-scale PHP web applications; Figure 11 describes the data set in more detail. These programs were chosen because code injection defect reports were available for them via an existing program analysis; building on such an analysis helps to demonstrate the applicability of our decision procedure. More specifically, we ran our analysis on bug reports that we were able to reproduce using Wassermann and Su's original tool and for which we could easily generate regular language constraints. Their analysis only finds *direct* defects, a term used by Wassermann and Su to refer to defects based on standard input variables, in three of their five programs; we restrict attention to three programs here.

The total program size is not directly indicative of our running time; instead, our execution time is related to the complexity of the violating path and thus of the constraints generated along it. Control flow and primitive string functions along that path contribute to the complexity of and the number of constraints and thus the complexity of the final constraint solving.

We conducted our experiments on a 2.5 GHz Core 2 Duo machine with a 6 megabyte L2 cache and 4 gigabytes of RAM. Figure 12 lists our results applying our decision procedure to produce user inputs (testcases) for 17 separate reported SQL injection defects; each row corresponds to a PHP source file within the listed application. In 16 of the 17 cases, the analysis took less than one second. The `secure` testcase took multiple minutes because of the structure of the generated constraints and the size of the manipulated finite state machines. In our prototype large string constants are explicitly represented and tracked through state machine transformations. More efficient use of the intermediate NFAs (e.g., by applying NFA minimization techniques) might improve performance in those cases.

We have implemented our decision procedure as a stand-alone utility in the style of a theorem prover [13, 14] or SAT solver [33, 48]. The source code is publicly available.[2] Our decision procedure was able to solve all of the regular language constraints generated by our simple symbolic execution approach. The ease of constructing an analysis that could query our decision procedure, the relative efficiency of finding solutions, and the possibility of solving either part or all of the graph depending on the needs of the client analysis argue strongly that our analysis could be used in practice.

5. Related Work

Language Theory and String Analyses. There has been extensive theoretical work on language equations; Kunc provides an overview [28]. Work in this area has typically focused on complexity bounds and decidability results. Bala [2] defines the *Regular Language Matching (RLM)* problem, a generalization of the Regular Matching Assignments (RMA) problem that allows both subset and superset constraints. Bala uses a construct called the *R-profile automaton* to show that solving RLM requires exponential space. Our decision procedure supports a different set of operations (e.g., we do not allow Kleene $\star$ on variables).

[2] `http://www.cs.virginia.edu/~ph4u/dprle/`

Vulnerability	|FG|	$\|C\|$	T_S
eve edit	58	29	0.32
utopia login	295	16	0.052
profile	855	16	0.006
styles	597	156	0.65
comm	994	102	0.26
warp cxapp	620	10	0.054
ax_help	610	4	0.010
usr_reg	608	10	0.53
ax_ed	630	10	0.063
cart_shop	856	31	0.17
req_redir	640	41	0.43
secure	648	81	577.0
a_cont	606	10	0.057
usr_prf	740	66	0.22
xw_mn	698	387	0.50
castvote	710	10	0.052
pay_nfo	628	10	0.18

Figure 12. Experimental results. For each of the SQL code injection vulnerabilities above, our tool was able to generate string values for input variables that led to the defect. |FG| represents the number of basic blocks in the code; $|C|$ represent the number of constraints produced by the symbolic execution step; and T_S represents the total time spent solving constraints, in seconds.

Christensen et al. first proposed a string analysis that soundly overapproximates string variables using regular languages [11]. Our implementation's constraint generator is based on an analysis by Wassermann and Su [43]. They extend Minamide's grammar-based analysis [32]. It statically models string values using context-free grammars, and detects potential database queries for which user input may change the intended syntactic structure of the query. In its original form, neither Wassermann and Su nor Minamide's analysis can generate example inputs.

In recent work, Wassermann et al. show that many common string operations can be reversed using finite state transducers (FSTs) [45]. They use this method to generate inputs for SQL injection vulnerabilities in a concolic testing setup. Their algorithm is incomplete, however, and cannot be used to soundly rule out infeasible program paths. Yu et al. solve string constraints [49] for forward symbolic execution, using approximations ("widening automata") for non-monotonic operations, such as string replacement, to guarantee termination. Their approach has recently been extended to handle symbolic length constraints through the construction of length automata [50]. The techniques proposed by Wasserman et al. and Yu et al. are potentially compatible with our implementation, and we propose investigating this in future work.

Bjørner et al. present a decision procedure for several common string operations by reduction to existing SMT theories [7], fixing the string lengths in a separate step. They show that the addition of a replace function makes the theory undecidable. In concurrent work, we show that bounded context-free language constraints can be solved efficiently by direct conversion to SAT [25]. Both approaches deal with individual string assignments. The algorithm presented here, in contrast, deals with languages rather than individual strings, and does not require (or reason about) string length bounds.

The Use of Decision Procedures. Decision procedures have long been a fixture of program analyses. Typically a decision procedure

handles queries over a certain *theory*, such as linear arithmetic, uninterpreted functions, boolean satisfiability [33, 48], pointer equality [34, 39], or bitwise operations and vectors [9, 15]. Nelson and Oppen presented a framework for allowing decision procedures to cooperate, forming an *automated theorem prover* to handle queries that span multiple theories and include first-order logic connectives [36]. In general, the *satisfiability modulo theories* problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. A number of SMT solvers, such as CVC [40] and Z3 [13], are available.

SLAM [5] and BLAST [20] are well-known examples of program analyses that make heavy use of external decision procedures: both are software model checkers that were originally written to call upon the Simplify theorem prover [14] to compute the effects of a concrete statement on an abstract model. This process, called predicate abstraction, is typically performed using decision procedures [29] and has led to new work in automated theorem proving [3]. SLAM has also made use of an explicit alias analysis decision procedure to improve performance [1]. BLAST uses proof-generating decision procedures to certify the results of model checking [19], just as they are used by proof-carrying code to certify code safety [35].

Another recent example is the EXE project [10], which combines symbolic execution and constraint solving [26] to generate user inputs that lead to defects. EXE has special handling for bit arrays and scalar values, and our work addresses an orthogonal problem. While a decision procedure for vectors might be used to model strings, merely reasoning about indexed accesses to strings of characters would not allow a program analysis to handle the high-level regular-expression checks present in many string-using programs. Our decision procedure fills this semantic gap.

Godefroid et al. [16] use the SAGE architecture to perform guided random input generation (similar to previous work on random testcase generation by the same authors [17, 18]). It uses a grammar specification for valid program inputs rather than generating arbitrary input strings. This allows the analysis to reach beyond the program's input validation stages. Independent work by Majumdar and Xu [30] is similar to that of Godefroid et al.; CESE also uses symbolic execution to find inputs that are in the language of a grammar specification. All of these projects could benefit from our decision procedure for strings and regular expressions when performing symbolic execution, which requires decision procedures for strongest-postcondition calculations as well as ruling out infeasible paths.

6. Conclusion

Many program analyses and testing frameworks deal with string variables that are manipulated by high-level operations such as regular expression matching and concatenation. These operations are difficult to reason about because they are not well-supported by current decision procedures and constraint solvers. This is of particular concern for analyses that use constraint solving techniques to generate, for instance, program inputs.

We present a decision procedure that solves systems of equations over regular language variables. We formally define the *Regular Matching Assignments* problem and a subclass, the *Concatenation-Intersection* problem. We then provide algorithms for both problems, together with a mechanized, machine-checkable proof for the core concat_intersect procedure, which we use inductively to solve the more general RMA problem. We also describe the theoretical space complexity of our algorithms.

We evaluate the utility and efficiency of our decision procedure empirically by generating constraints for 17 previously-reported SQL-injection vulnerabilities. In all cases, we were able to find feasible user input languages; in 16 of the 17 we were able to do so in under one second. The relative efficiency of our algorithm and the ease of adapting an existing analysis to use it suggest that our decision procedure is practical.

We assert that the precise modeling of string variables will be of critical importance to the next generation of program analyses and testing frameworks, especially for web applications. We formally defined a relevant problem and presented a decision procedure for it. The core of our decision procedure is mechanically verified, the procedure addresses a relevant and useful problem, and our experimental results indicate that even a prototype implementation can be practical and efficient.

References

[1] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *Static Analysis Symposium*, pages 230–246, 2002.

[2] S. Bala. Regular language matching and other decidable cases of the satisfiability problem for constraints between regular open terms. In *STACS*, pages 596–607, 2004.

[3] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification*, pages 457–461, 2004.

[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.

[5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, May 2001.

[6] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[7] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[8] British Broadcasting Corporation. UN's website breached by hackers. In `http://news.bbc.co.uk/2/hi/technology/6943385.stm`, Aug. 2007.

[9] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 358–372, 2007.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Computer and Communications Security*, pages 322–335, 2006.

[11] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *International Symposium on Static Analysis*, pages 1–18, 2003.

[12] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[13] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer-Aided Verification*, pages 519–531, 2007.

[16] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[18] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz

testing. In *Network Distributed Security Symposium (NDSS)*, 2008.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification*, pages 526–538, 2002.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[21] K. J. Higgins. Cross-site scripting: attackers' new favorite flaw. Technical report, `http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_1`, Sept. 2006.

[22] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *International Conference on Automated Software Engineering*, pages 73–82, 2007.

[23] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation*, pages 38–47, 2005.

[24] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Symposium on Security and Privacy*, pages 258–263, 2006.

[25] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory.

[26] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symposium*, pages 218–234, 2005.

[27] M. Kunc. The power of commuting with finite sets of words. *Theory Comput. Syst.*, 40(4):521–551, 2007.

[28] M. Kunc. What do we know about language equations? In *Developments in Language Theory*, pages 23–27, 2007.

[29] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[30] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, pages 134–143, 2007.

[31] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2005.

[32] Y. Minamide. Static approximation of dynamically generated web pages. In *International Conference on the World Wide Web*, pages 432–441, 2005.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[34] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Principles of Programming Languages*, pages 327–338, 2007.

[35] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[36] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[37] A. Salomaa, K. Salomaa, and S. Yu. State complexity of combined operations. *Theor. Comput. Sci.*, 383(2-3):140–152, 2007.

[38] K. Sen. Race directed random testing of concurrent programs. In *Programming Language Design and Implementation*, pages 11–21, 2008.

[39] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, pages 32–41, 1996.

[40] A. Stump, C. W. Barrett, and D. L. Dill. Cvc: A cooperating validity checker. In *Computer Aided Verification*, pages 500–504, 2002.

[41] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Principles of Programming Languages*, pages 372–382, 2006.

[42] P. Thiemann. Grammar-based analysis of string expressions. In *Workshop on Types in Languages Design and Implementation*, pages 59–70, New York, NY, USA, 2005. ACM.

[43] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Programming Language Design and Implementation*, pages 32–41, 2007.

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

[45] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *International Symposium on Software testing and analysis*, pages 249–260, 2008.

[46] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[47] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Usenix Security Symposium*, pages 179–192, July 2006.

[48] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.

[49] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, pages 306–324, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.