Generating String Inputs using Constrained Symbolic Execution

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

in Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Science

by

Pieter Hooimeijer

May 2008

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Master of Science

Computer Science

Pieter Hooimeijer

Approved:

Westley R. Weimer (Adviser)

David Evans (Chair)

Grigori R. Humphreys

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

May 2008

Abstract

The most commonly reported security vulnerabilities are related to cross-site scripting and SQL command injection. Both types of attack affect web applications that produce structured output such as SQL, XML, and HTML. Attacks that exploit these vulnerabilities are popular because web applications are ubiquitous, contain potentially valuable information, and are easily accessed remotely.

We present an efficient static analysis that detects SQL injection vulnerabilities and the execution paths that lead to them. Our algorithm is based on recent work that models sets of string values using context-free grammars. We extend this analysis so that its output—a set of structured bug reports—can be used to generate symbolic constraints over string variables. Our algorithm then solves these constraints, yielding a full set of attack inputs for each vulnerability that is detected.

The output of our algorithm is significantly easier to verify than the bug reports of the original analysis; we support this claim with anecdotal evidence. We also allow the developer to place restrictions on the path generation algorithm. This feature, in the spirit of bounded software model checking, allows our algorithm to automatically rule out certain classes of false positives.

We empirically evaluate the scalability of our implementation by applying it to a previously-published set of error reports. Our prototype successfully finds violating inputs for 17 out of the 22 bug reports that we were able to reproduce.

Contents

List of Figures

Chapter 1

Introduction and Background

In this thesis, we present an efficient static analysis that finds string inputs for applications that handle structured data, such as XML, SQL, and source code. Given the observable output of such an application, we use constraint solving techniques to "reverse engineer" feasible inputs that yield the given output. We apply this technique to automatically find feasible for SQL injection attacks.

1.1 SQL Command Injection

Web applications continue to increase in popularity and are commonly used for on-line business, government and collaborative purposes. These applications typically follow a tiered architecture. Important state information is stored in a back-end database, while the application layer (in conjunction with the user's web browser) make up the user interface to this data. Vulnerabilities in the application layer can provide a malicious user with indirect access to the back-end, jeopardizing potentially important information such as user accounts and credit card numbers.

A common attack of this type exploits SQL command injection vulnerabilities, in which arbitrary user-supplied commands are passed to the database. These vulnerabilities are quite prevalent in practice. In 2006, they made up 14% of reported vulnerabilities and were thus the second most commonly-reported security threat [18]. In addition, some speculate that these vulnerabilities are exploited much more frequently than other types of vulnerabilities, because highly-accessible web applications often hold highly-valuable information [36].

The crux of the problem is that web applications often communicate with back-end databases by passing strings that represent queries. Specialized functions exist to sanitize untrusted user data or present it to the back-end database in an uninterpreted manner, but such functions are rarely used correctly along all program paths. Such mistakes are particularly prevalent in scripting languages such as PHP, which lend themselves to the use of strings as a default data representation. The end result is that some paths through the application pass unchecked user data to the database.

Recently, much attention has been devoted to static techniques that detect and report potential injection vulnerabilities (e.g., [21, 26, 40]). Attacks remain prevalent [5], 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 [19, 39], and time is particularly relevant for security vulnerabilities.

We developed an analysis that augments a popular static injection vulnerability detector [38] by automatically computing symbolic constraints on attack inputs. Using these constraints, our tool generates sample attack inputs that can be used as a test case for the detected vulnerability. Our tool is based on an analysis that models string values using context-free grammars [29], and we extend this grammar to associate individual productions with path predicates [7,20,33] and program location information. Each predicate corresponds to the full set of conditions under which the final database query might contain substrings generated by a grammar production.

We use the annotated grammar to enumerate strings that correspond to injection attacks, considering only derivations with internally consistent path predicates. Each derivation yields a coarse path through the program's control flow graph. While generating strings, we concurrently search for full program paths that properly enclose the coarse paths obtained from the string generation step. We validate each full path using forward symbolic execution. If the path feasibly leads to the vulnerability, we combine the symbolic execution constraints and the path predicates to find sample input variables.

The main contributions of this thesis are:

- A constraint-generation and constraint-solving approach for string-valued variables and primitive string functions. Our algorithm extends previous work [6, 8] by including support for the

```
1  $username = 'admin';
2  $newsid = $_POST['posted_newsid'];
3  if ($get_posted_username != 0) {
4    $username = $_POST['posted_username'];
5  }
6  if (!preg_match('/[\d]+$/', $newsid)) {
7    unp_msgBox('Invalid article news ID.');
8    exit;
9  }
10 if ($get_posted_username != 0 && ereg('[\'"]', $username)) {
11   unp_msgBox('Invalid username.');
12   exit;
13 }
14 $newsid = "nid_" . $newsid ;
15 $mynews = $DB->query("SELECT * FROM 'news' WHERE poster='$username'");
16 $idnews = $DB->query("SELECT * FROM 'news' WHERE newsid='$newsid'");
```

Figure 1.1: SQL code injection vulnerability example adapted from Utopia News Pro. The `$_POST` mapping holds untrusted user-submitted data.

regular-expression handling functions that are common in web application code. It produces user input values that demonstrate a given vulnerability.

- An interleaved algorithm that extends static SQL code injection defect reports. The algorithm explores the state space of unsafe queries and the state space of program execution paths in tandem, using combined constraint information to rapidly prune infeasible regions. The algorithm produces test inputs and execution paths for real defects.

- An algorithm for producing annotated context-free grammars that describe the values of string variables at program points. We extend previous work [29, 38] by including path predicates and location information in the grammars. We formalize the construction of this grammar. The grammars are sound in that they derive a superset of the strings that can occur.

1.2 Motivating Example

Figure 1.1 shows a code fragment adapted from 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 detect a potential vulnerability on lines 15 and 16. The query string on line

15 is constructed from constant string fragments as well as the `$username` variable. In some cases, that variable is under user control and in some cases, the variable is not checked for unescaped quotes. However, those circumstances are never obtained on the same run through the code, and thus a report about line 15 is a false positive.

The vulnerability on line 16 is quite real. The check on line 6 is designed to limit `$newsid` to numbers: `[\d]+` is a regular expression for a non-empty sequence of consecutive digits. The delimiters `$` and `^` match the end and the beginning of the string respectively, but the check on line 6 is missing the `^`. Thus it is possible that the query sent on line 16 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 8.

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 [19, 39]. 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
```

In addition, we would like to provide a complete path through the program associated with the testcase, as well as 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 2 and 6, helping the developer locate potential causes of the error [2]. 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 point out the problem by generating a feasible test case that includes string input values and a viable execution path through the program that triggers the vulnerability.

In the rest of this thesis we present and evaluate an automatic algorithm that rules out some false positives, such as the one on line 15, and generates user inputs for real vulnerabilities, such as the one on line 16. In Chapter 2 we present our algorithm, including annotated grammar construction (Section 2.2), string enumeration (Section 2.3), path enumeration (Section 2.4), and constraint-solving input generation (Section 2.5). We present experimental results in Chapter 3, and follow up by placing our work and related work in context in Chapter 4.

Chapter 2

Proposed Algorithm

We propose an algorithm that extends the defect reports generated by a specific type of static analysis introduced by Minamide [29]. The defect reports generated by Minamide's algorithm consist of a context-free grammar for each string variable at a location of interest. This grammar represents an overapproximation of all possible values that this variable might take at runtime. Our algorithm generates a similar model of the program, but adds a mapping from the context-free grammar back to the program source code. This mapping is then used to automatically find specific values for input variables that could cause the reported defect. Under certain conditions, our algorithm can determine that the original defect report is actually a false positive. Our algorithm relies on external decision procedures and models of string-manipulating functions.

2.1 Overview

Intuitively, the algorithm searches the space of all possible string values that can reach the location and all paths that generate each string. It restricts attention to strings that violate a policy. For each such violating string, the algorithm searches for execution paths that could feasibly lead to that value. Once a path is found, constraint information from two static analyses is used to find values for input variables.

The inputs to our algorithm are (1) a control-flow graph F that represents a program in static single assignment form; (2) the program location l of a single string, such as the argument to a

```
find_inputs(flowgraph $F$, location $l$, policy $V$) =
let $G$ : grammar = annotated_grammar$(F,l)$ $\cap$ regular language $V$
let $U$ : variables = user_inputs$(F)$
foreach string $s \in L(G)$ do
    let $P_1 \dots P_k$ : productions = productions of $G \rightarrow^* s$
    let $C$ : constraints = constraints of $P_1 \dots P_k$ in $G$
    if consistent$(C)$ then
        let $l_1 \dots l_n$ : locations = locations of $P_1 \dots P_k$ in $G$
        foreach path $p \subseteq F$ that reaches $l$ via $l_1 \dots l_n$ in $F$ do
            let $\sigma$ : constraints = symbolic_execution$(F,p)$
            if consistent$(\sigma \cup C \cup \{l = s\})$ then
                return solve_constraints$(U, \sigma \cup C \cup \{l = s\})$
        end for
end for
return $(F,l,V)$ is a false positive
```

Figure 2.1: High-level pseudocode to find user inputs for the program F that lead to the string value at location l taking on a value in V. F is a flowgraph in SSA form, l is program location that represents a string variable, and V is a regular expression policy.

database query function that represents the potential defect; and (3) the *policy* V, a regular expression describing illegal string arguments to location l. High-level pseudocode for the algorithm is given in Figure 2.1; it proceeds as follows.

1. We first construct an annotated grammar G' that soundly approximates the string values possible at location l in the program. In other words, the context-free language $L(G')$ of G' is a superset of the values that could flow into location l at runtime. We intersect that grammar with the regular expression policy V. This results in G, a new grammar that describes the set of possible values for l that violate the policy. We formalize the construction algorithm annotated_grammar in Section 2.2.

   The resulting grammar G contains per-production control flow information and per-production constraints on program variables. These annotations are used in the steps that follow.

2. If the language $L(G)$ is empty, then we report that no defect was found. Otherwise, we enumerate strings s in the language of the grammar, keeping track of the productions used in each derivation. From this set of productions, we obtain a set of constraints on program vari-

ables and a set of program locations that must be visited. Note that $L(G)$ may be infinite, so the string enumeration algorithm allows the user to place restrictions on how strings are produced (e.g., by bounding the maximum length). Section 2.3 describes the string enumeration algorithm in detail.

If the string enumeration step yields a valid string (i.e., the set of conditions C is internally consistent), then we enumerate paths through the program that reach the location of interest. This is done via a backwards reachability analysis from the location l to the start of the program. Each path should (1) result in the desired string value s at location l; (2) visit each of the locations $l_1 \dots l_n$ (but not necessarily in order); and (3) be internally consistent. Conditions (1) and (3) are verified using symbolic execution. The backwards reachability analysis ensures that condition (2) is met for each generated path. The path generation step is explained in Section 2.4.

3. Once we have a concrete, consistent path through the program that reaches the location l with a concrete string value s that violates the policy V, we find values for the user inputs U. The symbolic execution state includes information about variable assignments and conditional guards. Together with the requirement that $l = s$, that symbolic state forms an underconstrained system of equations with respect to U. We use off-the-shelf techniques (e.g., integer linear programming) to find values for integer variables. For string variables, we show that many common string operations can be reversed symbolically using operations over finite automata. This final step is described in Section 2.5.

4. If the intersected grammar is empty or no feasible strings or paths exist, or if we exhaust all such strings or paths (up to some limit, if specified by the user), then we treat the original defect report as a false positive. Our algorithm is relatively sound with respect to the underlying decision procedures and string function models (*cf.* Section 2.6).

In the rest of this chapter we describe the steps of our algorithm.

2.2 Constraint-Annotated Grammars

The grammar generation step is based on the algorithms presented by Minamide [29] and Wassermann and Su [38]. We extend these algorithms by propagating location and path predicate information; we formalize the new algorithm in Section 2.2.1. The grammar generated by our algorithm is an approximate model of the program, and is necessarily imprecise in certain cases. In Section 2.2.2 we characterize where context-free grammar based analyses incur false positives.

2.2.1 Grammar Construction

Given a program F and a string expression at location l, we wish to construct a context-free grammar G such that $L(G)$ is a superset (i.e., a conservative overapproximation) of the values that l can take at runtime. In addition, we wish to annotate each production in the grammar with a set of constraints over program variables and a set of program locations. A number of precise algorithms exist for extracting such grammars (e.g. [9,37]), but we require a close connection between the structure of the grammar G and the structure of the program F. We thus extend previous work [29,38] to include constraint and location annotations. We also present the first formalization of the grammar construction itself; previous approaches have assumed the grammar and focused on formalizing finite state transducers to model string functions [29] or taint propagation in CFG-FSA intersection [38].

We produce the annotated grammar by recursively processing the abstract syntax tree for the program. Since the resulting grammar will contain productions associated with the values that variables can take on at runtime, we introduce a mapping M from program variables to grammar nonterminals to track where in the grammar each variable is described. Each string variable at a location is associated with exactly one nonterminal in the grammar. Not every grammar nonterminal necessarily represents a program variable, since some nonterminals may be used, for instance, to model loops and other control flow constructs. We write P for a set of constraints, each of which is a boolean-valued program expression. The grammar G itself is a set of annotated productions $A \underset{C}{\rightarrow} [Aa]^*$ where A is a nonterminal, C is a set of constraints and $[Aa]^*$ is a possibly-empty sequence

$$\dfrac{\begin{array}{c} X = \text{fresh}(x) \\ M' = M[x \mapsto X] \\ G' = G \cup \{X \xrightarrow[P]{} \texttt{string}\} \end{array}}{M,P,G \vdash x := \texttt{'string'} \ : \ M',P,G'}\ \textsf{literal} \qquad \dfrac{\begin{array}{c} M,P,G \vdash c_1 \ : \ M',P',G' \\ M',P',G' \vdash c_2 \ : \ M'',P'',G'' \end{array}}{M,P,G \vdash c_1;c_2 \ : \ M'',P'',G''}\ \textsf{seq}$$

$$\dfrac{\begin{array}{c} X_0 = \text{fresh}(x_0) \\ M' = M[x_0 \mapsto X_0] \\ G' = G \cup \{X_0 \xrightarrow[P]{} M(x_1)M(x_2)\} \end{array}}{M,P,G \vdash x_0 := x_1 \ . \ x_2 \ : \ M',P,G'}\ \textsf{concat} \qquad \dfrac{\begin{array}{c} X_0 = \text{fresh}(x_0) \\ M' = M[x_0 \mapsto X_0] \\ G' = G \cup \{X_0 \xrightarrow[P]{} M(x_1)\} \end{array}}{M,P,G \vdash x_0 := x_1 \ : \ M',P,G'}\ \textsf{assign}$$

$$\dfrac{\begin{array}{c} X_1 = \text{fresh}(x) \quad X_2 = \text{fresh}(x) \\ M' = M[x \mapsto X_1] \\ G' = G[M(x)] \cap \mathcal{L}(\texttt{re}) \\ G'' = G \cup \{X_1 \xrightarrow[P]{} S_{G'}\} \\ M',P,G'' \vdash c \ : \ M'',P',G^{(3)} \\ M^{(3)} = M''[x \mapsto X_2] \\ G^{(4)} = G^{(3)} \cup \{X_2 \xrightarrow[\textsf{true}]{} M(x) \ ; \ X_2 \xrightarrow[\textsf{true}]{} M''(x)\} \end{array}}{M,P,G \vdash \textsf{assume}(\textsf{ereg}(\texttt{re},x)) \ \textsf{do} \ c : M^{(3)},P',G^{(4)}}\ \textsf{match}$$

$$\dfrac{\begin{array}{c} M,P \wedge b_1,G \vdash c_1 \ : \ M',P',G' \\ \{x_1,\ldots,x_n\} = \text{assigned}(c_1) \\ X_i = \text{fresh}(x_i) \\ G'' = \bigcup\{X_i \xrightarrow[P]{} M'(x_i) \ ; \ X_i \xrightarrow[P]{} M(x_i)\} \\ G^{(3)} = G'' \cup G'[M(x_1) \mapsto X_1]\ldots[M(x_n) \mapsto X_n] \\ M'' = M'[x_1 \mapsto X_1]\ldots[x_n \mapsto X_n] \end{array}}{M,P,G \vdash \textsf{while} \ b_1 \ \textsf{do} \ c_1 \ : M'',P \vee P' \wedge \neg b_1,G^{(3)}}\ \textsf{while}$$

$$\dfrac{\begin{array}{c} M,P \wedge b_1,G \vdash c_1 \ : \ M',P',G' \\ M,P \wedge \neg b_1,G \vdash c_2 \ : \ M'',P'',G'' \\ G^{(3)},M^{(3)} = \text{rename}_{(M,M')}(G'',M'') \\ G^{(4)} = G' \cup G^{(3)} \end{array}}{M,P,G \vdash \textsf{if} \ b_1 \ \textsf{then} \ c_1 \ \textsf{else} \ c_2 \ : M^{(3)},P' \vee P'',G^{(4)}}\ \textsf{if}$$

Figure 2.2: Inference rules for construction of the annotated grammar. M is a mapping from program variables to grammar nonterminals, P is a path predicate that is true just before the current expression, and G is the annotated grammar. We write $\text{rename}_{(M',M'')}(G,M)$ for the alpha renaming of nonterminals in M and G to avoid clashes with mappings in M'' relative to M'.

of terminals and nonterminals.

The interpretation of a constrained production $A \underset{C}{\rightarrow} [Aa]^*$ is that the production may not take place along paths in which C is false. More formally, for a string variable x, if the values of variables along a particular program path p make C_i false for all derivations $M(x) \rightarrow \ldots \underset{C_i}{\rightarrow} [Aa]^* \rightarrow$ "`string`", then x cannot take on the value "`string`" along that path p. Recall that the grammar is a sound approximation (i.e., a superset) of the values that string variables can take on. However, if we can use the constraint information to rule out all derivations of a given string value, then that string value is guaranteed not to occur at runtime.

We introduce a judgment:

$$M, P, G \vdash e : M', P', G'$$

It can be read as, "given a mapping M, a set of constraints P that describe the path leading to e, and a grammar G, we process expression e to obtain the new mapping M', the new path predicates P', and the new grammar G'."

Figure 2.2 gives the syntax-directed inference rules for this judgment. The literal rule for string constants is an indicative base case: a new nonterminal X is created, a new grammar production is created that maps X to the string constant, and the mapping M is updated to reference X. Since the annotated grammar models string values explicitly and stores scalar values in constraints, we are only concerned with assignments involving string variables or string operations.

The assign and concat rules map string assignment and concatenation at the expression level to their equivalent operations at the grammar level. As before, a new nonterminal X_0 is created to represent x_0 on the left-hand side of the assignment. The right-hand side of the production is obtained using lookups in the mapping M. For instance, if we have $M(x_1) = X_1$ and $M(x_2) = X_2$, then the concat rule yields the new production $X_0 \underset{P}{\rightarrow} X_1 X_2$.

The match rule is an indicative example of a built-in string function. To keep the inference rules syntax-directed, we assume that the abstract syntax has been transformed so that predicates involving primitive string functions have been transformed to assume statements. For example, `if (ereg(a,b) && p)` { `c` } becomes `if (p)` { `assume(ereg(a,b)) do c` }. Intuitively, this allows us to distinguish between string-based branches (which affect the grammar itself) and branches that do not depend on string variables (which are handled by the if rule).

The assume block structure modifies the grammar to reflect the fact that variable b must match regular expression a within the block. Analogously, when the assumption is false, such as in the corresponding `else` block, the variable is restricted to the inverse of the language described by a. $G[M(x)]$ represents the context-free grammar generated by nonterminal $M(x)$. We write $\mathcal{L}(\mathtt{re})$ for the finite state automaton associated with the given regular expression; we intersect the grammar $G[M(x)]$ with that finite state automaton using the algorithms presented by Minamide [29]. Note that this intersection algorithm must be aware of the exact semantics of the `ereg` function; each built-in string matching function must be modeled separately.

The productions from X_2 have predicate true because the variables that it maps to ($M(x)$ and $M''(x)$) have been given the correct predicates inductively. These two options represent two possible executions paths. $M(x)$ represents the values of x before the assume block, while $M''(x)$ represents the values of x after executing c while assuming that $\mathsf{ereg}(\mathtt{re},x)$ is true. Note that $M^{(3)}(x)$ returns X_2; previous mappings for x are overwritten.

Similarly, the if rule deals with any branches that do not depend on string values. It generates the grammar for each branch separately. These grammars are then unioned after variable renaming to prevent clashes. Note that there are several other ways in which two context-free grammars might be unioned. We choose this method because we require that the grammar mimic the control and data flow of the program. Analogously, we require that the path predicate environment P be updated to represent either possible execution path ($P' \vee P''$). This is necessary to account for any side effects that c_1 and c_2 might cause.

Like the assume construct, the while rule modifies the grammar inferred from its body c_1. The set assigned(c_1) contains all string variables that appear on the left-hand side of an assignment in c_1.[1] Intuitively, we want to make any string operations in c_1 cyclic because they can happen repeatedly as the loop executes multiple times. Algorithmically, this is similar to a conversion to

[1]Computing assigned(c_1) requires that any pointer variables be treated conservatively. If a particular reference variable could point to several locations, then we can update the grammar for those locations separately. In this case, we would allow each location to take either its new value or its existing value.

```
$str = 'aa';
while(unknown()) {
   $str = str_replace('a', 'aa', $str);
}
```

Figure 2.3: Example of code that cannot be modeled precisely; the language for `$str` after the loop is $\{a^{2^n} \mid n \in \mathbb{N}\}$, which is not context-free.

SSA form. In this context, the fresh nonterminal X_i represents the Φ-function for a given program variable x_i, selecting between x_i's original value $M(x_i)$ or its value after being modified in the loop body $M'(x_i)$.

The two choices in the branching productions $X_i \underset{P}{\rightarrow} M'(x_i); X_i \underset{P}{\rightarrow} M(x_i)$ correspond to taking the while loop and not taking the while loop. The notation $G[Y_i \mapsto Z_i]$ stands for the grammar G with each instance of Y_i replaced with Z_i on the right-hand side of all productions. We use such a replacement for each variable in assigned(c_1), to make each of these variables' subgrammars cyclic. We expect the final path predicate to be $P \vee P' \wedge \neg b$, which corresponds to the loop executing or not. Here we expect P' to be representative of any number of executions of c.

In Figure 2.2, we omit the propagation of program location information for clarity of presentation. Our implementation associates such a location with each production, through a process that is equivalent to the propagation of predicates through P. Whenever a production is added to the grammar, we associate with it the location of the expression under consideration.

Given a particular variable of interest x from a defect report we find the nonterminal $M(x)$ and make it the start symbol of the grammar G. Finally, on line 2 of Figure 2.1 we intersect the resulting grammar with a regular expression V representing invalid strings that violate the safety policy (e.g., strings associated with code injection attacks). It is possible to adapt context-free language reachability to intersect a CFG and a regular expression without constructing the intermediate push-down automaton [28]. Wassermann and Su [38] extend that algorithm to track their own annotations, and we further extend it to propagate our grammar annotations.

2.2.2 Approximating String Operations

The context-free grammar construct also allows common string operations, such as substring replacement, to be modeled. These operations are elided from Figure 2.2, but are generally similar to the match rule in that they use specific operations over the grammar (e.g., regular intersection, image under a finite state transducer). Minamide [29] demonstrates how most string operations can be modeled.

Some string operations cannot be modeled precisely under all circumstances. The substring replacement function `str_replace` is an example of this. Figure 2.3 shows that, if used in a loop, `str_replace` can produce results that are not context-free. Each iteration of the loop on lines 3-5 doubles the length of `$str`. Hence, after the loop exits, the set of possible values for `$str` includes all strings of the form a^{2^n} for any n. This language is not context-free, so in general we are forced to approximate. One last-resort possibility is to allow `$str` to take any value (Σ^*); in practice our algorithm would yield the approximation a^* for this example. In either case, the grammar may produce strings that the program does not, potentially causing the analysis to produce false positives.

2.3 Violating Strings

Given the annotated grammar G, we seek to find a string $s \in L(G)$ and a path through the program along which l takes on the value s. We could enumerate paths and then strings associated with those paths, or enumerate strings and then paths associated with those strings. Given the annotated grammar, however, the violating strings are both more constrained than, and more constraining than, the program paths. We demonstrate this using several examples in Section 2.3.1; we then provide a string enumeration algorithm in Section 2.3.2.

2.3.1 Constraining Paths based on Strings

If the language of the grammar is empty, then the set of values that l could take is disjoint from the policy V. Even if $L(G)$ is non-empty, however, not every string it contains corresponds to a feasible

```
1  $str = $_POST['posted_string'];
2  if (!preg_match('/^[a-z]+$/', $str)) {
3    exit;
4  }
5  while (!preg_match('/00.*11/', $str) ) {
6    $str = "0" . $str . "1" ;
7  }
8  $DB->query($str);
```

Figure 2.4: Example of imprecision in grammar generation. $L(G)$ for `$str` at location 8 will include `000a111`. In fact, only `00a11` is possible.

```
1  $str = $_POST['posted_string'];
2  if ($x > $y) $str = "0" . $str ;
3  if (!preg_match('/^0?[a-z]+$/', $str)) {
4    exit;
5  }
6  if ($x <= $y) $str = "0" . $str;
7  $DB->query($str);
```

Figure 2.5: Example of imprecision in grammar generation. $L(G)$ for `$str` at location 7 will include `00a`, `0a` and `a`. However, `a` is infeasible at runtime.

program state. This is because G is computed as an approximation in general (see Section 2.2). Loops in the program and approximately-modeled string functions are common sources of imprecision in practice. For example, consider the code in Figure 2.4. The check on line 2 restricts `$str` to exactly `[a-z]+`. The while loop on line 5 brackets `$str` with 0 and 1 until there are at least two 0's and two 1's at the ends. The grammar G for `$str` at location 8 will generate the strings `a`, `0a1`, `00a11`, `000a111`, and so on, even though only `00a11` is feasible for the program. The constraints associated with the annotated grammar will allow us to rule out `a` and `0a1`, since they do not satisfy the negation of the loop guard from line 5.

Correlated conditionals involving integer variables will also cause a imprecision, as in Figure 2.5. The grammar G for `$str` at location 7 will generate the strings `a`, `0a`, `00a`, and so on, even though the checks on lines 2 and 6 exactly cover all cases and thus `a` is infeasible. In this case, the constraints associated with the annotated grammar will rule out `a`. The use of constraints usefully limits the output of $L(G)$, but it should be noted that it cannot rule out all infeasible strings in general.

```
enumerate_string(grammar $G$, queue $Q$) =
 let $\langle x$ : term/nonterm list, $p$ : production list$\rangle$ = take from $Q$
 if $x$ contains only terminals then
    if lazy consistency $\wedge$ $\neg$consistent($p$) then
       yield enumerate_string($G, Q$)
    else
       return $\langle x, p, Q\rangle$
 let $y$ : (term/nonterm list $\times$ production list) list = [ ]
 for $i = 1$ to length($x$) do
    if $x_i$ is a nonterminal then
       foreach $rhs_j$ such that $(x_i \rightarrow rhs_j) \in F$ do
          if lazy consistency $\vee$ consistent($p \wedge (x_i \rightarrow rhs_j)$) then
             add $[(x_1 \ldots x_{i-1}\ rhs_j\ x_{i+1} \ldots x_n), (rhs_j\ p)]$ to $y$
       end for
 end for
 if random order is desired then
    randomize $y$
 yield enumerate_string($G, Q \circ y$)
```

Figure 2.6: Annotated grammar enumeration algorithm. The algorithm is lazy and maintains a worklist Q that tracks its current state. Each call either returns a string s in $L(G)$ or *yields*, allowing string enumeration and path enumeration to be interleaved.

2.3.2 String Enumeration and Consistency Checking

We must thus enumerate the strings s in $L(G)$ and check them for feasibility. Figure 2.6 gives pseudocode for our lazy string enumeration algorithm. Each element in the worklist Q is a list of terminals and nonterminals paired with a list of production indices. The terminals and nonterminals represent an intermediate step in the derivation of a string; the production indices indicate which productions have been used in that derivation so far. We begin processing with Q containing $[S]$ and an empty production index list. If the next element x in the worklist is composed entirely of terminals, it is a string and we return it. Otherwise, we find all nonterminals x_i contained in x and all productions $x_i \rightarrow rhs_j$ in the grammar. For each such right-hand-side, we add a new element to the worklist consisting of the current partial derivation with x_i replaced by rhs_j. As an example, if G contains the productions $S \rightarrow aSb \mid \varepsilon$ and $T \rightarrow t$ and SdT is the next element in the worklist, we will add $aSbdT$, dT and Sdt to the worklist.

This is analogous to a breadth-first exploration of the grammar, because new elements corresponding to the use of one production for each nonterminal are appended to the worklist. If $L(G)$ is non-empty, each call to this algorithm will terminate with a string in $L(G)$, or an indication that the worklist is empty. An empty worklist indicates that $L(G)$ is finite and has been completely enumerated. If the algorithm cannot generate a feasible string in one step, then it *yields* before calling itself recursively. This allows the caller to interrupt the string enumeration algorithm and interleave calls to the path enumeration algorithm.

Since the annotated grammar G comes equipped with constraints on each production, we can avoid enumerating strings that correspond to infeasible paths. We can perform an *eager* check of consistency at each production application or a final *lazy* check of consistency before returning each final string. For example, consider a grammar G containing only the productions $S \xrightarrow[x=0]{} aSb$, $S \xrightarrow[x=1]{} \varepsilon$ and $T \xrightarrow[x=0]{} t$, $T \xrightarrow[x=1]{} TS$. If $x = 0$ and $x = 1$ are the constraints, we might eagerly cut off the derivation $ST \rightarrow aSbT \rightarrow aSbt \rightarrow \ldots$ and stop exploring the portion of $L(G)$ that derives from $aSbt$. Eager consistency checking involves multiple calls to a decision procedure per generated string, but has the potential to rule out entire sublanguages of violating strings.

Finally, it may be desirable to explore the grammar in a random order. The grammars generated by the analysis in Section 2.2 often have many productions for a given nonterminal (i.e., have a high "branching factor"). For example, consider $S \rightarrow a \mid b \mid \ldots \mid z \mid 0 \mid 1 \mid \ldots \mid 9$. If a final string containing a digit is desired, an in-order traversal will necessarily consider many unhelpful productions before considering a useful one. These "branching" nodes often result from regular expression ranges, such as `[a-z0-9]`. From the perspective of the final safety policy, the different elements in a range often matter much less than the various corner-case characters. Exploring the range sequentially is therefore often suboptimal.

2.4 Program Path Enumeration

We validate a given string $s \in L(G)$ by enumerating paths p through the program F that lead to location l. Our basic algorithm is again a worklist breadth-first search, but the location constraints provided by the grammar annotations severely limit the search space and help to ensure termination.

```
enumerate_paths(program $F$, location set $r$, queue $Q$) =
  let $v$ : flowgraph node list = take from $Q$
  let $v_1$ : flowgraph node = first element of $x$
  if $v_1$ is the start node then
    return $\langle$reverse($v$), $Q\rangle$
  let $y$ : flowgraph node list list = [ ]
  foreach $u$ such that Predecessor($v_1, u$) do
    if $r \cap$ (Predecessor$^*(u) \cup v) = r$ then
      add $(u\ v)$ to $y$
  end for
  if random order desired then
    randomize $y$
  yield enumerate_paths($F, l, Q \circ y$)
```

Figure 2.7: Program path enumeration algorithm. The algorithm is lazy and maintains a worklist Q to track its state. It traces backwards using predecessor edges in the flowgraph F. The set r contains required locations that must occur on the path. Predecessor* is the reflexive, transitive closure of the Predecessor relation on control flow graph nodes.

Back edges and loops related to string variables are modeled in the grammar G. For example, in Figure 2.4, for the string $s =$ `000a111`, three uses of the $S \rightarrow 0S1$ production require that the path include location 6 exactly three times. Once a particular string $s \in L(G)$ has been selected, we need not re-examine all looping paths through the program: we will either be constrained to a particular number of iterations through a loop, or that loop will not involve the string variable s.

Figure 2.7 gives pseudocode for our flowgraph path enumerator. Our algorithm is lazy and returns a new path each time it is called. The worklist Q is initialized to $[l]$, the singleton list containing the program location with the vulnerability. The algorithm traces backwards from l to the start node. The set r contains locations that are required to occur in the final path; those locations are drawn from the annotated grammar productions used to produce the violating string under consideration in Section 2.3.

We compute the transitive reflexive closure of the Predecessor relation in advance. We use this transitive reachability information to avoid exploring a transition if it would prevent us from fulfilling the obligations in r. The enumeration is breadth-first, and thus each call on a non-degenerate flowgraph will either return a path from the start node to l, or indicate that all paths conforming to r have been explored. As with string enumeration, it may be desirable to explore the flowgraph in a random order. Finally, we present r as a set of locations for simplicity of exposition; it can be directly lifted to a multiset or an ordered list in practice.

Once we have a path p to location l associated with a string $s \in L(G)$, we can perform symbolic execution. Because the path is of finite length and does not branch, symbolic execution always terminates. The resulting symbolic execution state will contain a symbolic valuation for program variables as well as a set of guards or assumptions associated with conditionals and loops. Each guard is a constraint over program variables. For example, a path through the code in Figure 2.5 that takes the conditional on line 2 and the conditional on line 6 will produce the guard set $\{x > y, x \leq y\}$. An automated theorem prover or other decision procedure can rule out such infeasible paths. If the path is feasible, we use the constraints from the grammar, the string s, and the symbolic state to generate values for user input variables.

2.5 Generating Values for Input Variables

Once we have discovered a viable finite path p through the program F that reaches the error location l with a violating string value $s \in V$, we need only determine values for input variables that exhibit a particular vulnerability. We will use the constraint information we have from the grammar productions used in $G \rightarrow^* s$ as well as the symbolic state associated with symbolically executing p. The system as a whole is typically underconstrained, leaving an amount of freedom in the final input values. It may also be overconstrained if the consistency check on line 11 of Figure 2.1 fails to detect an inconsistency. In practice this could happen if the inconsistency is based solely on string operations. In that case, we would detect the inconsistency during the constraint solving steps described in this chapter; such an inconsistency would result in an empty language solution for some string variable.

The task of generating inputs that force a program to take a particular path is a special case of the larger problem of automatic test-case generation. Recently, a number of projects have focused on using symbolic execution to generate such inputs [3, 13, 14, 23]. Constraints involving integer variables and even integer arrays with random accesses are well-understood (e.g., [6]). Most such scalar constraints are linear (e.g,. `$sub_action = 255` or `$pref` $\neq$ `0`) and can be addressed using integer linear programming. We do not propose any new algorithms for finding scalar input values and instead build on previous work in that area.

We focus on generating values for *string* input variables. Strings are particularly important for our example of SQL code injection vulnerabilities for web services: almost all user-supplied variables are freeform strings that come from HTTP GET and POST requests. Much of the relevant program logic involves string manipulation and checking. In one of our testcases, the `warp` program, 62% (567/902) of all `while` and `if` predicates (by static count) directly involved a string value. Reasoning about strings is of critical importance to finding informative input values within our target domain.

2.5.1 Algorithm Outline

In the sections that follow, we describe the steps required to find values for input variables of interest. This corresponds to the solve_constraints invocation on line 12 of Figure 2.1. Given a specific string value (the violating query) and a path that leads to it, we essentially want to reverse any operations along that path. We proceed as follows:

1. To determine the order in which to process the various symbolic constraints, we use the symbolic execution state to create a dependency graph (Section 2.5.2).

2. In Section 2.5.3, we present the *forward processing rules* used to eliminate nodes from the dependency graph. These rules apply as long as there are no cyclic dependencies in the graph.

3. In Section 2.5.4, we show that the forward processing rules do not always apply. We show that the dependency graph may have implicit cycles when two program variables are first concatenated and then filtered. We present the concat_intersect algorithm to solve a basic case of such a cycle. Of interest is the fact that dependency cycles may have several disjunctive solutions. In other words, we may have to explore several possible assignments of languages to nodes.

$$
\begin{array}{rcll}
c & ::= & \mathsf{True} & \\
  & | & \mathsf{False} & \\
  & | & \mathsf{String}(s) & \text{equal to literal string value } s \\
  & | & \mathsf{Eq}(n_i) & \text{equal to } n_i \\
  & | & \mathsf{NotEq}(n_i) & \text{not equal to } n_i \\
  & | & \mathsf{Concat}(n_i, n_j) & \text{equal to concatenation of } n_i \text{ and } n_j \\
  & | & \mathsf{Fun}(f, n_i, \ldots, n_j) & \text{built-in string function}
\end{array}
$$

Figure 2.8: High-level constraint language. Each dependency graph node is associated with a set of high-level constraints; the constraints induce edges between the nodes.

4. We then provide a more general algorithm that puts the previous two steps together. The algorithm (Section 2.5.5) uses a worklist to explore disjunctive solutions if they occur. The algorithm has several possible outcomes. If there does not exist any solution for some particular node (i.e., it is assigned the empty language for all possible assignments to other nodes), then the system was overconstrained and no solution is possible. Otherwise the algorithm results in a valid mapping from nodes to regular languages. We use this mapping to generate specific string values for variables of interest.

2.5.2 Dependency Graph Generation

Symbolic execution generates constraints for assignments and guarded control flow. For example, along the path 1-2-3-6-9 in Figure 1.1, we would expect these constraints:

```
username = 'admin'
newsid = index(POST, 'posted_newsid')
default_user = true
preg_match('/[\d]+$/', newsid) = false
```

The final constraint is particularly indicative: any string generated for `newsid` must end with at least one digit.

Our approach to constraint solving follows the equality DAG approach found in cooperating decision procedures [30]. Each program variable or string constant (or abstract location reported by an alias analysis, see e.g. [10]) is associated with a node in the DAG. Each node has an initially-empty set of *high-level constraints*. The language of high-level constraints is shown in Figure 2.8.

$$\frac{n \text{ is fresh}}{\vdash \text{"}\texttt{str}\text{"} : n, \{\langle n, \mathsf{String}(\text{"}\texttt{str}\text{"})\rangle\}} \qquad \frac{n \text{ is the node of } \mathit{variable}}{\vdash \mathit{variable} : n, \{\}}$$

$$\frac{}{\vdash \mathsf{false} : n_F, \{\}} \qquad \frac{}{\vdash \mathsf{true} : n_T, \{\}}$$

$$\frac{\vdash e_1 : n_1, C_1 \quad \vdash e_2 : n_2, C_2}{\vdash e_1 = e_2 : n_T, C_1 \cup C_2 \cup \{\langle n_2, \mathsf{Eq}(n_1)\rangle\}} \qquad \frac{\vdash e_1 : n_1, C_1 \quad \vdash e_2 : n_2, C_2}{\vdash e_1 \neq e_2 : n_T, C_1 \cup C_2 \cup \{\langle n_2, \mathsf{Neq}(n_1)\rangle\}}$$

$$\frac{\vdash e_1 : n_1, C_1 \quad \vdash e_2 : n_2, C_2 \quad n \text{ is fresh}}{\vdash e_1 \;.\; e_2 : n, C_1 \cup C_2 \cup \{\langle n, \mathsf{Concat}(n_1, n_2)\rangle\}} \qquad \frac{\vdash e_1 : n_1, C_1 \quad \ldots \quad \vdash e_k : n_k, C_k \quad n \text{ is fresh}}{\vdash f(e_1, \ldots, e_k) : n, \bigcup C_i \cup \{\langle n, \mathsf{Fun}(f, n_1, \ldots, n_k)\rangle\}}$$

Figure 2.9: Constraint generation rules. We generate a dependency graph by processing the symbolic state for a given path by recursive descent. Judgments take the form $\vdash e : n, C$, where e is a symbolic state expression, n is the associated graph node, and C collects a set of node-constraint pairs. This yields a subgraph for each symbolic constraint; the full dependency graph consists of the union of these sets.

Beyond the $\mathsf{Fun}(f, \ldots)$ constraint, which is used to model primitive string functions, the high-level constraints all have the expected meanings. In addition, unique nodes n_T and n_F are annotated with the True and False constraints, respectively.

We recursively process program expression constraints to obtain the nodes associated with expressions and a set of high-level constraints. C is the set of node-constraint pairs. If $\langle n, c\rangle \in C$ then node n has constraint c. Figure 2.9 gives the constraint generation rules. New nodes in the equality DAG are created to represent intermediate values, such as the result of a string concatenation. We apply the high-level constraint-generation rules to all of the symbolic execution constraints. Any unmodeled expression is given a fresh node after its subexpressions are examined.

We merge nodes with respect to the equivalence classes induced by Eq constraints. If $\langle n_i, \mathsf{Eq}(n_j)\rangle \in C$, we merge nodes n_i and n_j. This can be implemented efficiently using a disjoint-set data structure and a union-find algorithm. We then remove equality constraints from consideration and associate with every node a finite-state machine. The language of the FSM for node n contains all of the string values that n may take on at runtime. We write $F(n)$ for the FSM at node n and $S(n)$

$$\begin{array}{lll}
\langle n, \mathsf{String}(s)\rangle & \Rightarrow & F(n) \leftarrow F(n) \cap \mathsf{FSM\_of\_string}(s) \\
\langle n, \mathsf{NotEq}(n_i)\rangle & \Rightarrow & F(n) \leftarrow F(n) \cap \mathsf{Neg}(F(n_i)) \\
 & & \\
\langle n, \mathsf{Fun}(ereg, n_i, n_j)\rangle & \Rightarrow & F(n_j) \leftarrow \begin{cases} F(n_j) \cap \mathsf{FSM\_of\_reg}(S(n_i)) & \text{if } n = n_T \\ F(n_j) \cap \mathsf{Neg}(\mathsf{FSM\_of\_reg}(S(n_i))) & \text{if } n = n_F \end{cases} \\
 & & \\
\langle n, \mathsf{Concat}(n_i, n_j)\rangle & \Rightarrow & F(n) \leftarrow F(n) \cap \mathsf{Append}(F(n_i), F(n_j))
\end{array}$$

Figure 2.10: Forward constraint processing rules; these can be solved using FSM intersection. The Concat constraints may require additional processing if combined with other constraints on the same node.

for an arbitrary string in $L(F(n))$. Each FSM is initially set to accept Σ^*.

2.5.3 Forward Constraint Solving Rules

We now use the dependency graph to solve the string variable constraints. Most constraints update some $F(n)$ by intersecting it with their own FSM and thus restrict the set of possible string values. For these constraints, it suffices to sort the graph in constraint dependency order. For example, if $\langle n, \mathsf{NotEq}(n_i)\rangle \in C$ we will process n_i before processing n. Since FSM intersection is monotonic, this process always terminates. In language theory an infinite descending sequence of regular expression intersection is possible: $\Sigma^* \cap 1\Sigma^* \cap 11\Sigma^* \dots$ is an example. Because these constraints model a finite program path, however, there are only finitely many intersected regular languages based on the finite set of constraints and nodes.

Figure 2.10 gives the high-level constraint processing rules. We refer to these as *forward* processing rules because they follow the order induced by topologically sorting the dependency graph. Each rule updates $F(n)$ by intersecting it with another FSM. We write $\mathsf{Neg}(F)$ for the function that returns an FSM that accepts all strings not in $L(F)$. We write $\mathsf{Append}(F_1, F_2)$ for the function that returns an FSM that accepts all strings xy with $x \in L(F_1)$ and $y \in L(F_2)$. We write $\mathsf{FSM\_of\_reg}$ for the finite state machine associated with a regular expression.

The rule for $\mathsf{Fun}(ereg, n_i, n_j)$ is indicative of our handling of built-in string functions. The `ereg` function (and the `preg_match` function, etc.) returns a non-zero value if its second argument matches the regular expression given by the first argument. We first check to see if the node associated with that constraint is n_T or n_F, and thus whether the function's return value is believed to be true or false. If it is true, we find a string $S(n_i) \in L(n_i)$. This string describes a regular language, so we construct its machine $F(S(n_i))$ and intersect it with $F(n_j)$. If the constraint is false, we compute $\overline{F(S(n_i))} \cap F(n_j)$; in other words we negate the language described by $S(n_i)$ before intersecting. In practice, n_i is always a singleton string constant. We model many other string-handling functions that are variations on these themes (for example, `eregi` behaves as `ereg` but performs case-insensitive matching).

2.5.4 Finding Disjunctive Solutions

The forward constraint processing rules defined in Figure 2.10 are not sufficient to correctly process all dependency graphs. If a constraint $\langle n, \mathsf{Concat}(n_i, n_j)\rangle$ is combined with an intersection constraint (such as $\langle n, \mathsf{NotEq}(n_k)\rangle$), then the intersection constraint may propagate *backwards* through the dependency graph. In other words, the intersection constraint on n may also restrict the possible values for n_i and n_j.

Figure 2.11 provides a concrete example that features a concatenation (line 9) followed by a string filter function (lines 10–12). For ease of presentation, we will simplify the actual constraint graph by limiting it to two types of edges: $\cap$ and $\circ$. A $\cap$-edge indicates that the target node's language should be constrained by the source node's through regular intersection. A pair of $\circ$-edges indicates that the target node should be constrained by the concatenation of the graphically rightmost source node to the graphically leftmost source node. Note that each node can be the target of at most one $\circ$-edge pair, and any number of $\cap$-edges. For Figure 2.11, the actual graph would include nodes n_T and n_F, and Fun edges to model the `ereg` and `eregi` constraints introduced on lines 3, 6, and 10.

In this context, we perform the intersections for n_a and n_b first, eliminating nodes t_1 and t_2 from future consideration. Node n_c then depends on the concatenation edges and on t_3 by intersection. At this stage, we note that the intersection $n_c \cap t_3$ also restricts the possible values of n_a and n_b, even though there is no explicit edge in that direction. This restriction of n_a and n_b yields several solutions: input variable a can take on values `opp` or `opppp`, and variable b can be `pq` or `pppq`. In

```
// user inputs $a and $b
// can have any value
if(!ereg('o(pp)+', $a)) {
  exit;
}
if(!ereg('p*q', $b)) {
  exit;
}
$c = $a . $b;
if(!eregi('oppppq', $c)) {
  exit;
}
```

$$\begin{tikzcd}
t_1 \arrow[r, "\cap"] & n_a \arrow[dr, bend left] & & n_b \arrow[dl, bend right] & t_2 \arrow[l, "\cap"'] \\
 & & n_c \arrow[u, phantom, "\circ"] & t_3 \arrow[l, "\cap"'] &
\end{tikzcd}$$

$F(t_1) = \mathsf{FSM\_of\_reg}(o(pp)^+)$
$F(t_2) = \mathsf{FSM\_of\_reg}(p^*q)$
$F(t_3) = \mathsf{FSM\_of\_reg}((o|O)(p|P)^5(q|Q))$

Figure 2.11: Example: code that generates disjunctive solutions (left) together with a graphical representation of its dependency graph (right). The correct set of solutions for this dependency graph is $(n_a \equiv \mathrm{opp} \wedge n_b \equiv \mathrm{pppq}) \vee (n_a \equiv \mathrm{opppp} \wedge n_b \equiv \mathrm{pq})$.

addition, these solutions are strictly *disjunctive*; if we choose a to be `opppp` then b cannot be `pppq`.

The example in Figure 2.11 demonstrates that (1) constraints can propagate backward in the dependency graph; and (2) that we may encounter disjunctive solutions. We define the task of finding these disjunctive solutions as the *concatenation-intersection (CI)* problem. Formally, given three regular languages L_1, L_2, and L_3, the CI problem requires the set $S = \{\langle L'_1, L''_1\rangle, \ldots, \langle L'_n, L''_n\rangle\}$ that satisfies the following properties if $1 \leq i \leq n$:

1. L'_i and L''_i are regular.
2. $L'_i \subseteq L_1$ and $L''_i \subseteq L_2$
3. $L'_i \circ L''_i \subseteq (L_1 \circ L_2) \cap L_3$
4. $(L'_1 \circ L''_1) \cup \ldots \cup (L'_n \circ L''_n) = (L_1 \circ L_2) \cap L_3$
5. $\langle L'_g, L''_g\rangle, \langle L'_h, L''_h\rangle \in S \Rightarrow (L'_g = L'_h \Leftrightarrow L''_g = L''_h)$

The first three properties restrict the invididual entries of S. We need the solutions to consist of regular languages, since that is how we represent the set of possible values for each variable. Property two requires that each solution be a constraint on L_1 and L_2. In other words, a solution should not include strings that were not already available as "inputs" for the concatenation $L_1 \circ$

```
concat_intersect($L_1, L_2, L_3$) =
    Assume: $L_1, L_2, L_3$ have respective machines s.t. $M_i = \langle Q_i, \Sigma, \delta_i, s_i, f_i \rangle$
    let $L_4 = L_1 \circ L_2$ s.t. $M_4 = \langle Q_4 = (Q_1 \cup Q_2), \Sigma, \delta_4, s_1, f_2 \rangle$
    let $L_5 = L_4 \cap L_3$ s.t. $M_5 = \langle Q_5 = (Q_1 \cup Q_2) \times Q_3, \Sigma, \delta_5, (s_1, s_3), (f_2, f_3) \rangle$
    let $Q_{\text{lhs}} = \{(f_1, q') \mid q' \in Q_3\} \cap Q_5$
    let $Q_{\text{rhs}} = \{(s_2, q') \mid q' \in Q_3\} \cap Q_5$
    foreach $\langle q_1, q_2 \rangle \in Q_{\text{lhs}} \times Q_{\text{rhs}}$ s.t. $q_2 \in \delta_5(q_1, \varepsilon)$ do
        let $M'_1$ = induce_from_final($M_5, q_1$)
        let $M'_2$ = induce_from_start($M_5, q_2$)
        output $\langle M'_1, M'_2 \rangle$
    end for
```

Figure 2.12: Constraint solving for intersection across concatenation. The algorithm relies on basic operations over NFAs: concatenation using a single ε-transition (line 3) and the cross-product construction for intersection (line 4). The two induce functions are described in the text.

L_2. Analogously, property three defines what constitutes a valid solution $\langle L'_i, L''_i \rangle$: it should never introduce elements that are not in $(L_1 \circ L_2) \cap L_3$.

The last two properties restrict S itself. Property four states that S should contain all valid solutions. Property five, finally, requires that each solution be maximal and that S does not contain any duplicates. Intuitively, this requires that we merge any solutions pairs that are not actually disjoint. For example, if we have two satisfying solutions $s_1 = \langle L'_1, L''_1 \rangle$ and $s_2 = \langle L'_2, L''_2 \rangle$, and $L'_1 = L'_2$, then S should contain $\langle L'_1, L''_1 \cup L''_2 \rangle$. In this example, s_1 and s_2 are not disjoint solutions because they share an element, in this case the left-hand side. We show later that property five ensures that $|S|$ is finite.

Figure 2.12 provides high-level pseudocode for finding S. For simplicity, we assume that all NFAs have a single start state $s_i \in Q_i$ and a single final state $f_i \in Q_i$. To find S, we use the structure of the NFA M_5 (line 4) that recognizes $L_5 = (L_1 \circ L_2) \cap L_3$. The algorithm first constructs a machine for $L_4 = L_1 \circ L_2$ 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 $L_5 = L_4 \cap L_3 = (L_1 \circ L_2) \cap L_3$. The set of states Q_5 for this machine corresponds to tuples in the set $(Q_1 \cup Q_2) \times Q_3$, with $\delta_5((q_4, q_3), w) = \{(q, q') \mid q \in \delta_4(q_4, w) \text{ and } q' \in \delta_3(q_3, w)\}$.

Intuitively, machine M_5 consists of a left-hand side that represents L_1 intersected with some prefix part of L_3, and a right-hand side that consists of L_2 intersected with some suffix part of L_3.

The two sides are connected exclusively by zero or more ε-transitions, which represent the original concatenation of M_1 and M_2. In this context, if more than one disjunctive solution exists, then this is because the suffixes of the left-hand side overlap with the prefixes of the right-hand side.

We now treat each ε-transition between the left and right-hand machines as a potential solution. We isolate the two sides in lines 5 and 6. Q_{lhs} is the set of states in M_5 that corresponds to the original final state of M_1, and Q_{rhs} corresponds to the original start state of M_2. We process each ε-transition from Q_{lhs} to Q_{rhs} as follows:

- $\mathsf{induce\_from\_final}(M_5, q_1)$ (line 8) returns a copy of M_5 with q_1 marked as the only final state.
- $\mathsf{induce\_from\_start}(M_5, q_2)$ (line 9) 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 10, if either M_1' or M_2' describe the empty language, then we omit this solution pair from the output. We describe an example execution of this algorithm in Figure 2.15 (Section 2.5.5).

The output of this algorithm satisfies the five properties that define the correct solution set S (modulo duplicate solutions, which may occur if one of the input machines is not minimized). Each of the output pairs consists of two regular languages by definition, since each language is described by an NFA. A proof showing that the algorithm satisfies properties two, three, and four would proceed by induction on the structure of machines M_1, M_2, and M_3.

Every time we pick an ε-transition in M_5 (line 7), we restrict the strings that the resulting machine might accept, satisfying property three. For each such transition, the left-hand side corresponds to L_1 intersected with some subset of the prefixes of L_3, satisfying property two; the argument for the right-hand side is analogous. Property four holds because we use all live ε-transitions. Note that the number of ε-transitions of interest is bounded by the size of M_5, and must thus be finite.

The last property (which requires that S be minimal) is more difficult to show, because the algorithm depends on the structure of the input machines for L_1, L_2, and L_3. If these machines include redundant transitions and states, then this may result in redundant ε-transitions in M_5. This could be resolved by minimizing the inputs to the algorithm, or by minimizing the left and right-hand sides of M_5 separately before the output step.

2.5.5 General Constraint Graph Solving

We now combine the steps outlined in Sections 2.5.3 and 2.5.4 and provide a general algorithm for solving dependency graphs over strings. The algorithm keeps a worklist of partially-processed dependency graphs along with node-to-FSM mappings; this is necessary to handle disjunctive solutions. The initial worklist consists of the dependency graph that represents the full set of symbolic constraints. The the initial node-to-FSM mapping returns Σ^* for all nodes.

Figure 2.13 provides pseudocode for the algorithm. It first solves as many non-cyclic constraints as possible (line 3–9). The procedure sort_acyclic_nodes performs a topological sort of the graph up to any dependency cycles. The procedure process_acyclic_constraint uses the forward constraint processing rules described in Section 2.5.3 to process the constraints imposed by a single node; it may remove zero or more nodes from the graph after updating their corresponding FSMs. Optionally, as a performance consideration, we may choose to minimize any machines that were updated in the new mapping F'.

We will use the dependency graph in Figure 2.14 as a running example. This graph is partially processed; the assignments on the right-hand side represent the current mapping. The remaining graph is cyclic, and thus requires special handling. Note that the sort_acyclic_nodes procedure should avoid returning nodes that are cyclically dependent. We define a node n's *dependents* as the set of nodes that can be reached transitively as follows:

- Follow any outbound $\cap$-edges and $\circ$-edges.
- If the current node has an inbound $\circ$-edge-pair in addition to an inbound $\cap$-edge, then follow both $\circ$-edges backwards. Treat both $\circ$-edges as intersection constraints on their respective source nodes.

We define a *dependency cycle* as a set of nodes C that satisfies $\forall n \in C,\ C \subseteq \mathsf{dependents}(n)$. The cycle in Figure 2.14 is $\{n_a, n_e, n_b, n_f, n_c\}$. Note that n_d and n_g can each affect any element in this

```
solve_dependency_graph(queue $Q$, node set $S$) =
let $\langle G, F \rangle$ : graph $\times$ (node $\rightarrow$ FSM) = take from $Q$
let $N$ : node list = sort_acyclic_nodes($G$)
**for** $0 \leq i <$ length($N$) **do**
    let $n$ : node = $N[i]$
    let $\langle G', F' \rangle$ : graph $\times$ (node $\rightarrow$ FSM) = process_acyclic_constraint($n, G, F$)
    $F \leftarrow F'; G \leftarrow G'$
**end for**
let $C$ : node set = find_free_cycle($G$)
**if** $|C| > 0$ **then**
    let $\langle G', R \rangle$ : graph $\times$ (node $\rightarrow$ FSM) list = process_cycle($C, G, F$)
    $G \leftarrow G'; F \leftarrow$ head($R$)
    **foreach** $r \in$ tail ($R$) **do**
        add $\langle G, r \rangle$ to end of $Q$
**end if**
**if** $\forall s \in S.\ F[s] \neq \varnothing \wedge |G| = 0$ **then**
    **return** $F$
**else if** $\forall s \in S.\ F[s] \neq \varnothing \wedge |G| > 0$ **then**
    **return** solve_depencency_graph($\langle q, F \rangle$ :: $Q, S$)
**else if** $\exists s \in S$ **s.t.** $F[s] = \varnothing \wedge |Q| > 0$ **then**
    **return** solve_dependency_graph($Q, S$)
**else**
    **return** no inputs found
process_cycle(node set $C$, graph $G$, node $\rightarrow$ FSM $F$) =
let solutions : (node $\rightarrow$ FSM) list = [ ]
let $N$ : node set = maximal_concats($C, G$)
**for** $1 \leq i \leq |N|$ **do**
    let $D_i$ : (node $\rightarrow$ FSM) set = generalized_concat_intersect($n, C, G, F$)
**end for**
**foreach** $s = \langle d_1, \ldots, d_{|N|} \rangle \in D_1 \times \ldots \times D_{|N|}$ **do**
    **foreach** $c \in C$ **do**
        let $s' \subseteq s$ **s.t.** $d_i \in s' \Rightarrow d_i[c] \neq$ none
        **if** $\exists d_i, d_j \in s'$ **s.t.** $d_i[c] \not\equiv d_j[c]$ **then**
            $s \leftarrow$ adjust($s, s', c, G$)
    **end for**
    **if** $\neg(\exists c \in C,\ d_i \in s$ **s.t.** $d_i[c] = \varnothing)$ **then**
        add $d_1 \cup \ldots \cup d_{|N|}$ to solutions
    **end if**
**end for**
let $G'$ : graph = $G \setminus \{n \in G \mid$ dependents($n$) $= C\}$
**return** $\langle G', \text{solutions} \rangle$
```

Figure 2.13: Constraint solving algorithm for general dependency graphs over string variables. The algorithms uses a worklist of dependency graphs and node-to-FSM mappings. The graph represents the work that remains; successful termination occurs if all nodes are eliminated.

$$\begin{tikzcd}
 & n_a \arrow[dr] & & n_b \arrow[dl, "\circ"'] \arrow[dr] & & n_c \arrow[dl, "\circ"'] & \\
n_d \arrow[rr, "\cap"] & & n_e & & n_f & & n_g \arrow[ll, "\cap"']
\end{tikzcd}$$

$$n_a \equiv o(pp)^+$$
$$n_b \equiv p^*(qq)^+$$
$$n_c \equiv q^*r$$
$$n_d \equiv op^5q^*$$
$$n_g \equiv p^*q^4r$$

Figure 2.14: A dependency graph that exhibits an implicit dependency cycle. The right-hand side shows the initial constraints. In this case, n_b is affected by both intersection $n_d \cap n_e$ and $n_f \cap n_g$, making the two concatenations mutually dependent. The correct solution set for this graph includes all possible assignments to n_a and n_c for which there exists an assignment to n_b that simultaneously satisfies the constraints on n_e and n_f.

set, even though they are not part of the cycle itself. The procedure find_free_cycle (line 9) returns a cycle together with any nodes that are directly connected to a node in the cycle through a $\cap$-edge. We define a *maximal concat node* in a dependency cycle C as any node that has no outbound edges to another element in C. In Figure 2.14, n_e and n_f are maximal.

The process_cycle procedure takes as its parameters a set of nodes that form a dependency cycle, a graph, and the current node-to-FSM mapping. It returns an updated graph (with the appropriate nodes removed), and a list of zero or more alternative FSM mappings. For convenience, we assume that it returns at least one mapping (i.e., it returns a dummy mapping if no satisfying assignments are found).

To solve the cyclic constraints, we first find the *maximal concat nodes* in the cycle (line 26); recall that these are nodes that do not have any outbound edges to other nodes in the cycle. We then solve the separate constraints for each of those maximal nodes using a generalized version of the concat_intersect algorithm given in Figure 2.12. The generalized version handles arbitrary sequences of intersections and concatenations, rather than just a single one of each. The algorithm is analogous to the original, but requires tracking individual graph nodes (and the states belonging to each node's FSM) across multiple operations.

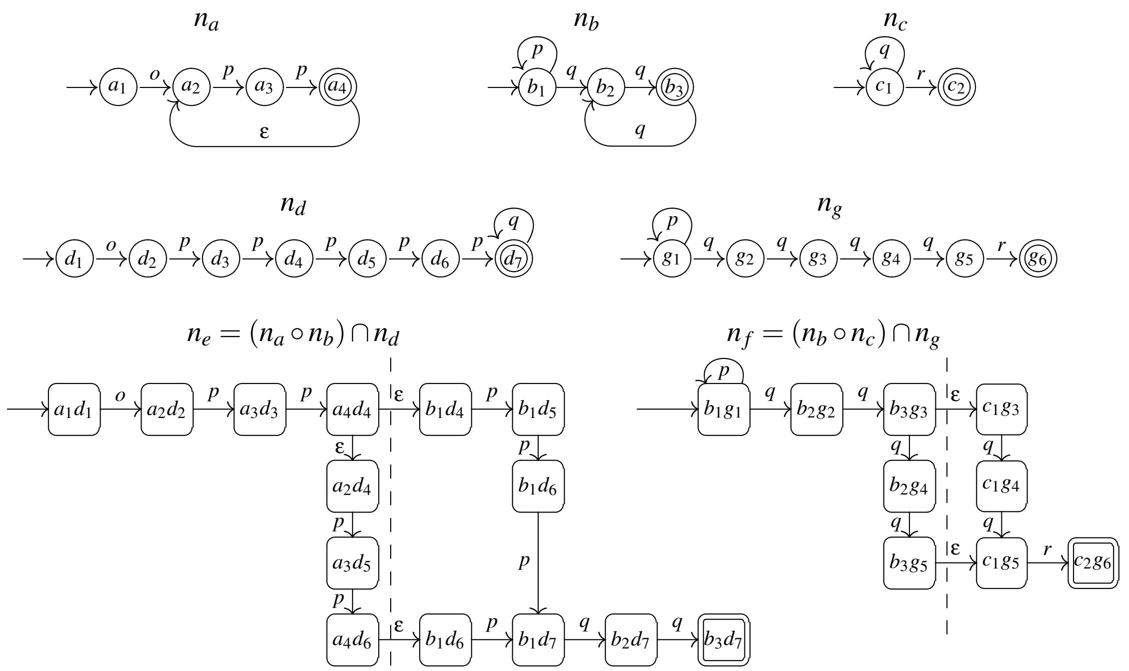
The maximal nodes in the example in Figure 2.14 are n_e and n_f. Figure 2.15 shows the automata associated with this dependency graph. The machine for n_e corresponds to the intermediate machine M_5 that is constructed by concat_intersect (Figure 2.12) on inputs $(F(n_a), F(n_b), F(n_c))$. The algorithm extracts two solutions from this machine: $n_a \equiv opp \wedge n_b \equiv pppqq$ and $n_a \equiv opppp \wedge n_b \equiv pqq$. These solutions correspond to the two ε-transitions that cross the dashed line in Figure 2.15. Analogously, the machine for n_g yields solutions $n_b \equiv p^*qq \wedge n_c \equiv qqr$ and $n_b \equiv p^*qqqq \wedge n_c \equiv r$.

So far we have solved for n_e and n_f individually, yielding two disjunctive solutions for each. If n_e and n_f were otherwise independent from each other, then we would return all four combinations as possible solutions. However, since n_b must *simultaneously* satisfy constraints on n_e and n_f, we must do additional filtering. In Figure 2.13, the separate solution pairs are be computed in lines 27–29. We then filter the solutions in lines 30–39, which consider all possible combinations of solutions.

Recall that each D_i is a set of possible solutions for some maximal concat node. We consider all combinations of solutions (line 30), and filter out any that yield impossible results. For each solution combination s and for each node c in the cycle, we check to see if the c is mapped to by more than one solution in s. For example, in Figure 2.14, nodes n_a would not be mapped in a solution for n_f, since n_f only affects n_b and n_c directly.

On line 33, the if-statement checks whether any of the regular languages for the given node disagree (i.e., are not equal). If such a disagreement exists, then adjust is called (line 34) to find a single satisfying assignment for the current node. The call adjust(s, s', c, G) computes $q = d_1[c] \cap \ldots \cap d_{|N|}[c]$, i.e. the greatest lower bound of the solutions for c. It returns a new combination of solution mappings $\langle d'_i, \ldots, d'_{|N|} \rangle$, where each $d'_i[c]$ is updated to map to q. In addition, each d'_i is updated so that any forward dependents of c use the new (possibly more restricted) mapping for c. Finally, we replace the original combination of solutions s with the adjusted version.

It is possible that adjust intersects two disjoint languages for a single node, yielding the empty language $\varnothing$ as the greatest lower bound. This means that the current combination of solution mappings is infeasible. In Figure 2.13, we check for this on line 36; if any node maps to $\varnothing$, then we reject the entire solution combination. If we were able to find a non-empty mapping for all variables (i.e., none of the intersections performed by adjust produced $\varnothing$), then we union the solutions. Note that, because of the calls to adjust, any variable that occurs in more than one map will have the same language across all maps.



n_a
n_b
n_c
n_d
n_g
$n_e = (n_a \circ n_b) \cap n_d$
$n_f = (n_b \circ n_c) \cap n_g$

Figure 2.15: Intermediate automata for solving Figure 2.14. The process_cycle procedure (Figure 2.13) calls concat_intersect (Figure 2.12) to find disjunctive solutions that satisfy the constraints on n_e and n_f separately. It then considers all combinations of these solutions and outputs the solution combinations that have matching machines n_b.

Recall that, if applied to the example in Figure 2.15, the algorithm considers all 2×2 combinations of solutions. The only node that participates in both concatenations is n_b. This rules out the solution for n_f that requires $n_b \equiv p^*qqqq \wedge n_c \equiv r$, since neither solution for n_e would allow n_b to contain strings ending in $qqqq$. In these two cases, a call to adjust would yield $b \equiv \varnothing$. The two other combinations, however, are viable:

1. Solution $n_a \equiv opp \wedge n_b \equiv pppqq$ for n_e and solution $n_b \equiv p^*qq \wedge n_c \equiv qqr$ for n_f. Intersecting the two assignments for n_b yields:

$$n_a \equiv opp \wedge n_b \equiv pppqq \wedge n_c \equiv qqr \wedge n_e \equiv opppppqq \wedge n_f \equiv pppqqqqr$$

2. Solution $n_a \equiv opppp \wedge n_b \equiv pqq$ for n_e and solution $n_b \equiv p^*qq \wedge n_c \equiv qqr$ for n_f. Intersecting

the two assignments for n_b yields:

$$n_a \equiv opppp \wedge n_b \equiv pqq \wedge n_c \equiv qqr \wedge n_e \equiv opppppqq \wedge n_f \equiv pqqqqr$$

These two solutions are returned by process_cycle in the form of two separate mappings together with the updated dependency graph. The main algorithm appends these solutions to its worklist and continues by either (1) terminating successfully (line 17); (2) continuing to solve the current graph (line 19); (3) attempting to solve a new graph from the worklist (line 21); or (4) terminating without having found a satisfying set of inputs (line 23).

The cycle solving in Figure 2.13 can be implemented efficiently by using the structure of the NFAs. Recall that we solve cycles by considering the solutions for *maximal* concat nodes, i.e., nodes that have no forward edges into the cycle. It is important to note that we can track states of finite automata across intersections and concatenations (both operations strictly increase the size of the resulting machine relative to the inputs). While constructing the machine for each maximal concat node, the algorithm can track the "origin" of each state. This is illustrated in Figure 2.15, where the machine states for n_e and n_f are labeled according to the corresponding states before the intersection.

The tracking allows for a mapping between dependency graph nodes and parts of the finite automata computed for the maximal nodes. For example, n_b would map to the right-hand side of the n_e machine and the left-hand side of the n_f machine. In this context, once a particular path through the larger machines has been selected, the adjust procedure can work on the single large machine rather than walking the dependency cycle repeatedly to update forward dependents.

When the algorithm terminates successfully, we choose an arbitrary $S(n)$ for each node that corresponds to a user input variable. Those concrete values become the testcase that demonstrates the vulnerability. Alternatively, we could convert the finite automata to regular expressions. This would provide developers with a slightly more general description of the vulnerability.

It is also possible to provide a precise slice of the program with respect to each user input variable by tracking which constraints cause our FSM models of user input values to change. We update our constraint-solving algorithm so that every update $F(n) \leftarrow F(n) \cap \ldots n' \ldots$ checks to see if the resulting finite state machine has changed, and if it has records that n depends on n'. When emitting the final value $S(n)$ for a user input variable we also report all of its transitive dependencies, as well as those for the SQL query string. In the worst case this is the same as a data dependency slice on those variables, but because it includes our modeled knowledge of primitive string operations it will exclude string operations that were present on the path but did not affect the final values. We leave computing these slices (and experimentally evaluating their utility) for future work.

2.6 Relative Soundness

Our algorithm is relatively sound with respect to the decision procedures and models of primitive string functions it employs. This relative soundness arises in three areas. First, the annotated grammar construction relies on a modeling of primitive string functions. Second, ruling out infeasible paths relies on decision procedures that can reason about the functions along those paths. Third, generating input based on constraints involving primitive string functions relies on a proper modeling of those primitive string functions; in practice this modeling corresponds to that used by the grammar construction.

If the decision procedures used are not sound, our algorithm will consider infeasible paths as feasible and will thus generate input values that do not demonstrate the vulnerability. However, precise decision procedures for arrays and bit-vectors are available (e.g., [12]); it would be reasonable to adapt them to strings or to translate queries about primitive string functions into their input language.

The steps of the algorithms listed in this chapter have the following possible outcomes:

- The annotated context-free grammar construction (Section 2.2) overapproximates the possible string values that might occur at runtime. If it does not show any violations (i.e., no user-controlled variables that violate the policy can reach a database statement), the the program must be safe. If a flaw is found, then it could either be real or a false positive.

- The string (Section 2.3) and path enumeration (Section 2.4) attempt to find paths that lead to a specific string value at a program point. This step depends on external procedures (i.e., a theorem prover), and can be made conservative by only considering paths that are *not* provably *in*feasible.

- The constraint solving algorithm (Section 2.5) is itself a decision procedure. If the symbolic constraints are computed over a path that has an impossible string constraint, then our algorithm will fail to find any inputs. Our algorithm is not complete, however, since we do not model all possible string operations. Dealing with unmodeled operations can be done conservatively (by rejecting any paths that have them) or optimistically (by setting the operands to Σ^*).

Chapter 3

Experiments

Our prototype implementation takes defect reports generated by Wassermann and Su's tool [38] and applies the algorithm in Figure 2.1. We use the Simplify automated theorem prover [11] to implement the consistent() subroutine that determines if sets of constraints are mutually satisfiable. Our implementation of symbolic_execution() includes McCarthy's treatment of memory updates, again using Simplify to decide aliasing queries. We used the lazy consistency option in our string enumeration algorithm (see Figure 2.6); this places a greater burden on our enumeration and scheduling code and less of a burden on the automated theorem prover.

Our experiments test the expressive power of our algorithm and measure its runtime efficiency. First, we experimentally validate that our algorithm can find string-valued inputs that lead to code injection vulnerabilities. Second, we verify that it does so in a reasonable amount of time. The experiments here were conducted using representative set of input parameters to our algorithm; we leave measuring the effects of parameter changes for future work.

3.1 Testcases

We ran our experiments on the data set used by Wassermann and Su [38], which consists of several large-scale PHP web applications; Figure 3.1 describes the data set in more detail. These programs were chosen because code injection defect reports were available for them. More specifically, we run our analysis on bug reports that we were able to reproduce using Wassermann and Su's original tool.

We eliminated tiger and e107 from the dataset. Both testcases are reported to have defects [38]. In the case of e107, the large code base made it difficult locate the single direct bug reported by Wassermann and Su. Tiger is reported to have only *indirect* vulnerabilities, which depend on non-standard input variables (e.g., results from one database query that are fed into the next). Analyzing this type of vulnerability requires finding all possible writes to a particular database location, and analyzing each for vulnerabilities. This is not possible with our current prototype.

Since our algorithm starts once the defect has been reported, the total program size is not directly indicative of running time; instead, our execution time is related to the complexity of violating path. Control flow and primitive string functions along that path contribute to the complexity of the annotated grammar, the complexity of enumerating strings, the number of paths to consider, and the number of constraints and thus the complexity of the final constraint solving.

3.2 Experimental Results and Discussion

Figure 3.2 lists our results for 22 separate defects; each row corresponds to a PHP source file within the listed application. The columns are categorized to match the main steps in the algorithm: *Grammar* for grammar generation; *String Enum* for the process of finding and verifying violating strings; *Path Enum* for finding paths associated with verified strings; and *Solve* for generating and solving the dependencies and finding actual inputs. Note that, in the prototype, string enumeration and path enumeration are interleaved, so the times recorded for these steps are not contiguous. The *T* column lists the total cumulative time for our algorithm (see Figure 2.1) including decision procedure calls and all subprocedures, and including I/O; all times are reported in seconds.

We conducted our experiments using a representative set of parameters. We used in-order path and string enumeration (i.e., no randomization); we evaluated string derivation consistency lazily (i.e., after generating an entire string rather than at each derivation step); and we interleaved each string generation step with 15 path enumeration steps. We bound string generation to strings of

Name	Version	Description	files Total	loc Total	Vulnerable
e107	0.7.5	content management	741	132,862	N/A
eve	1.0	activity tracker	8	905	3
tiger	1.0 beta 39	news management	30	6,701	0
utopia	1.3.0	news management	24	5,438	5
warp	1.2.1	content management	44	24,365	14

Figure 3.1: Description of the Wassermann and Su [38] data set. The *vulnerable* column lists the number of files in which the Wassermann and Su analysis found a potential SQL injection vulnerability; in our experiments we attempt to find inputs for the first vulnerability in each such file.

	Grammar		String Enum			Path Enum				Solve				
Vulnerability	$\|G\|$	T_G	$\|G'\|$	N_{SE}	T_{SE}	$\|FG\|$	N_{PE}	N_B	T_{PE}	$\|C\|$	N_S	T_S	T	V
eve edit	267	0	25	1	0	58	1	0	0	34	1	0.39	1	✓
member	992	55	29	1	0.01	200	1	0	0	75	–	0	55	
user	553	1	29	1	0	45	1	0	0	40	–	0	1	
utopia login	829	1	40	1	0	295	1	0	0	24	–	14	15	$\checkmark_{ni}$
profile	4746	7	29	1	0	855	1	0	0	24	–	1	30	$\checkmark_{ni}$
styles	5253	4	221	1	0.05	597	1	0	0	80	16	0	6	✓
users	45905	30	500	1	2.89	1015	1	104	0.09	79	–	0	37	
comm	20084	28	25	1	0.03	994	1	133	0.11	153	29	0.18	32	✓
warp cxapp	5968	19	25	1	0	620	1	0	0	18	–	678	697	$\checkmark_{ni}$
ax_help	6518	35	25	1	0	610	1	0	0	8	–	0	35	$\checkmark_{ni}$
ax_main	82325	134	25	1	0	1182	1	0	0	–	–	0	142	
usr_reg	10534	27	25	1	0	608	1	0	0	17	–	1	28	$\checkmark_{ni}$
cart_pay	21934	200	25	1	0.03	739	1	0	0.11	348	–	0.02	201	
ax_ed	6498	18	25	1	0	630	1	0	0	8	–	0	19	$\checkmark_{ni}$
cart_shop	16227	55	25	1	0.05	856	1	0	0.11	252	–	8	63	$\checkmark_{ni}$
req_redir	11118	37	25	1	0.02	640	8	0	0	40	–	251	289	$\checkmark_{ni}$
secure	11272	31	25	1	0.01	648	1	0	0.02	72	–	5	36	$\checkmark_{ni}$
a_cont	10560	39	25	1	0	606	1	0	0	7	–	1	40	$\checkmark_{ni}$
usr_prf	29682	141	25	1	0.03	740	1	0	0	99	–	10	151	$\checkmark_{ni}$
xw_mn	11355	50	25	1	0.01	698	1	0	0	443	49	0.01	51	✓
castvote	11363	43	25	1	0.01	710	1	9	0	31	–	0	44	$\checkmark_{ni}$
pay_nfo	15656	55	25	1	0.01	628	1	0	0	360	–	18	74	$\checkmark_{ni}$

Figure 3.2: 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. The $|G|$ column lists the number of productions in the annotated grammar *before* intersection with the policy; T_G lists the time taken to construct G and extracting subgrammars for individual variables of interest. $|G'|$ is the size of the grammar *after* intersection with the policy. N_{SE} is the number of strings that were generated from G', and T_{SE} was the total time spent in enumerate_strings. $|FG|$ is the size of the control flow graph of the files under consideration; N_{PE} is the number of full execution paths generated; N_B is the number of paths that were rejected because they had too many back edges; T_{PE} lists the total time spent in enumerate_paths. $|C|$ represent the number of constraints produced by the symbolic execution step; N_S is the number of input variables that were solved for. T_S represents the total time spent solving constraints. T lists the entire analysis time from start to finish; a check in the V column indicates that algorithm successfully found inputs ($\checkmark_{ni}$ is discussed in the text).

length at most three, and we traversed backward edges in the control flow graph at most once. We conducted our experiments on a 3 GHz dual core Xeon machine with a 4 megabyte cache and 32 gigabytes of RAM.

We were able to generate inputs for 17 of the 22 defects we considered; these testcases have a check mark in the V column. Our prototype had significant difficulty determining which variables should be considered user inputs, especially for those applications that use references (e.g., arrays) to store user inputs across include boundaries. This is a limitation of our implementation; our symbolic interpreter for PHP handles a strict subset of the language features. Xie and Aiken [40] outline some of the difficulties in analyzing PHP and dynamic scripting languages in general. An indicative example is the `extract` function[1], which injects each $(key, value)$ pair of an associative array into the local scope, possibly shadowing extant local variables. The `extract` function is used extensively in the Warp testcases.

We used two separate implementations of the constraint solving algorithm discussed in Section 2.5. Implementation A uses the built-in FSM procedures provided by Minamide's analysis, and analyzes the entire constraint graph. Implementation B attempts to find input variables before solving constraints, allowing it to skip any variables that are not of interest. Implementation B, while more efficient than A, supports fewer features because we do not use Minamide's built-in data structures.

Figure 3.2 shows the results of applying our algorithm to 22 separate testcases. We use Implementation B whenever possible, and revert to the alternate implementation whenever needed. A testcase marked ✓ indicates that Implementation B successfully found input variables and correct values for them. We use $\checkmark_{ni}$ ("no inputs") to indicate that we used the slower implementation to solve the entire constraint graph. A testcase with no check mark indicates that both implementations failed; in practice this means that the slower implementation failed to terminate.

The performance numbers show that the analysis time is dominated by the grammar generation step (column T_{G} in Figure 3.2). None of the testcases required the enumeration of more than one string (N_{SE}), although several testcases had intermediate paths rejected because they followed a

[1] http://us3.php.net/manual/en/function.extract.php

loop too many times (N_B). The strings generated for specific inputs tended to be limited in length; the most common occurrences were `'` and `'9`. This may explain why the constraint solving step is relatively fast; short strings correspond to manageable FSM sizes.

These results suggest that our technique scales to reasonably sized programs (thousands of control flow graph nodes; hundreds of string operations). SQL injection vulnerabilities lend themselves well to constraint solving because violating strings are easily characterized and, more importantly, short. Future work might reasonably adapt this technique to problems that require more complex constraints, such as code generation and static XML validation [37].

Chapter 4

Related Work

The detection of SQL injection vulnerabilities, both statically and at runtime, is a well-studied problem. Other related work focuses on automated testcase generation based on the output of, for instance, software model checkers. In this chapter we provide an overview of analyses and techniques that are similar to ours.

4.1 Dynamic Detection and Enforcement

There are many methods for dynamic detection of SQL injection vulnerabilities. Nguyen-Tuong *et al.* [31] propose a hardened PHP interpreter that tracks tainted data and prevents it from influencing database queries. Pietraszek and Berghe [32] take a similar approach, associating metadata with string values and processing values in a context-sensitive manner. Similarly, Halfond *et al.* [17] track “positive string taint,” which represents trusted data. This approach avoids false negatives in practice and imposes little runtime overhead. Su and Wassermann [35] model legitimate SQL queries using a grammar-based approach. They only allow query strings if its user inputs are properly contained at a particular level of the derivation.

While dynamic methods are beneficial, they impose at least some runtime overhead. Our method finds feasible user inputs that cause a statically-detected vulnerability to manifest. This is advantageous over dynamic tools because it eliminates runtime overhead and provides a concrete testcase for use in software development.

4.2 Static Detection Methods

The Pixy tool [21] uses data-flow analyses of PHP programs to detect cross-site scripting vulnerabilities. It tracks the flow of tainted data to sensitive sinks using memory, literal, and especially alias analyses. Pixy does not model possible string values, however, and suffers from a high false positive rate (near 50%). Our approach, in addition to using taint information, models actual string values. This allows us to rule out several classes of false positives, such as those caused by non-standard sanitization functions. We are also able to rule out some false positives through our string enumeration step, relative to the soundness of the decision procedures and string function models we use.

The PQL language [26] is a query language that depends on programmer input. PQL allows programmers to specify application-specific code patterns, such as vulnerable sinks and sanitization functions. Static and dynamic program checkers are then generated based on those manually-specified queries to detect potential errors. Because of its general approach, PQL is limited to high-level taint propagation checks, and cannot model actual string values the way our analysis does.

Xie and Aiken [40] model programs at the intrablock, intraprocedural, and inter-procedural level. This gives their analysis precision at the important intraprocedural level, while promoting scalability. Their method computes "summaries" of program behavior at various levels of granularity, and checks the final summary for SQL injection vulnerabilities. A notable difference between their analysis and our own is that their analysis requires programmer input to accurately predict the effects of regular expressions; we require no user input.

Our implementation is based on that of Wassermann and Su [38]. They extend Minamide's grammar-based analysis [29]. Wassermann and Su's analysis 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. We modify this analysis to make its output amenable to further automated processing, allowing our algorithm to search for actual execution paths and corresponding inputs. Our constraint solving algorithm is not tightly coupled with the original analysis by Wassermann and Su; it can be applied to any scheme that supplies a valid path through the code and a string value at a program point of interest.

4.3 Language Based Approaches

Language-level approaches, such as relating program-level strongly-typed classes to database schema [22, 27], effectively prevent the type of injection vulnerability that we model. Persistent object systems [1, 4] also obviate the need for potentially-dangerous database queries by handling data persistence under the hood at the object level. In general, however, these methods do not apply to existing code bases that do not use them. Moreover, using call-level queries-are-strings interfaces such as ODBC and JDBC remain the most common in practice [34].

Leijin and Meijer [24] propose a language-level solution, arguing that embedding domain-specific languages like SQL in higher-order typed languages can provide syntactic and type-system safeguards against dangerous or illegal queries. They claim that their Haskell representation of a scripted web program is easier to maintain and guarantees safety.

While we favor language-level techniques in the long-term, they do not help to correct defects in existing legacy systems. Our approach works on existing PHP code and with existing SQL injection attack detectors.

4.4 Finding Violating Inputs

Our work builds on past work that attempts to generate user inputs from constraints (e.g., [6]) or via bounded model checking (e.g., [8]). Recent work in this area has referred to this process as “whitebox fuzzing,” using program information (such as execution traces or static analysis) to guide otherwise random testing [14].

The EXE project [6] combines symbolic execution and constraint solving to generate user inputs that lead to defects. EXE has special handling for bit arrays and scalar values, and our work is similar to theirs in spirit. We assume the existence of such tools for handling scalar constraints, and focus on generating *string* input values instead. We provide algorithms for giving inputs in the presence of regular expression constraints rather than array-element-level constraints.

The DART project [15] provides an automatic unit testing framework by combining static and dynamic analyses. Given only the program (in source and executable form), DART compares symbolic executions with actual runs of the program. The framework then uses the symbolic state to direct further random testing (i.e., to generate inputs that will exercise different paths through the code). The framework tests for failed assertions and program crashes.

Although both DART and our analysis automatically generate testcases, the two have substantially different goals. DART generates no false positives; any bug that it finds corresponds to an actual failed execution. In general, however, a 'clean' testing run with DART does not guarantee program correctness. Conversely, our analysis can be used to show a program correct with regard to a policy. It may, however, generate false positives (in the form of inputs that do not exercise an actual bug).

Godefroid *et al.* [14] use the SAGE architecture to perform guided random input generation (similar to previous work by the same authors [15, 16]). 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. The goal here is similar to that of DART, and the same differences relative to our work apply. The CESE project, done independently by Majumdar and Xu [25], is similar to that of Godefroid *et al.*. CESE similarly uses symbolic execution to find inputs that are in the language of a grammar specification.

It is worth noting that general full-program symbolic execution cannot be used directly to solve the input-generation problem we are addressing. Without user-supplied loop invariants or special widening or join operators, symbolic execution does not generally terminate on programs containing loops. Our method of annotating the grammar with location information means that we can avoid enumerating infinite paths. In addition, the constraint solving approach we take to find input variables can be viewed as a join operator for an abstract interpretation system in which strings values are modeled as finite state machines.

Chapter 5

Conclusion

We present an efficient static analysis for finding execution paths and concrete values for user input variables that lead to SQL injection vulnerabilities. These input variables form a testcase that makes it more likely that conventional software engineering methodologies will address the problem.

Our algorithm starts by generating an annotated grammar that soundly approximates all strings values that can reach a given program point. Each production in this annotated grammar comes equipped with program location information and program constraints. We then enumerate strings in that grammar, taking only those strings whose grammar productions have consistent constraints. For each such string we enumerate possible paths through the control flow graph, limiting our search to those paths that visit the locations required by the string's grammar productions. We check each path for feasibility using forward symbolic execution. When we find a feasible path that leads to the vulnerability we solve the constraints from the grammar and the constraints from symbolic execution to obtain values for the user inputs. Our constraint solving works by iterative regular language intersection, with special handling for dependency cycles.

Our algorithm as a whole is relatively sound with respect to the decision procedures and primitive string function models that it employs. In practice, this means that the algorithm will generate only strings that lead to the vulnerability, provided that our model of the program is accurate. In our experiments, our algorithm found viable user inputs for 17 of 22 vulnerabilities, taking at most 12 minutes per vulnerability to do so.

Beyond our modeling of PHP primitive string functions, our algorithm is not specific to PHP or SQL injection vulnerabilities. Given decision procedures and models of the relevant string-manipulating functions, it can be used in broader contexts where values for user inputs are desired that cause a program to reach a known location with a given value.

Bibliography

[1] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, 1995.

[2] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[4] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 403–417, New York, NY, USA, 2003. ACM Press.

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

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

[7] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[8] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 73–82, New York, NY, USA, 2004. ACM.

[9] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *International Symposium on Static Analysis*, pages 1–18, 2003.

[10] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

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

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

[13] Patrice Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[14] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 9–11, 2008.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[16] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*, 2008.

[17] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT FSE*, pages 175–185, 2006.

[18] 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`, September 2006.

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

[20] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Programming Language Design and Implementation (PLDI)*, pages 38–47, New York, NY, USA, 2005. ACM Press.

[21] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[22] W. Keller. Mapping objects to tables – a pattern language. In *European Conference on Pattern Languages of Programming Conference*, 1997.

[23] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[24] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, New York, NY, USA, 1999. ACM Press.

[25] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ASE*, pages 134–143, 2007.

[26] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, 2005.

[27] Russell A. McClure and Ingolf H. Krüger. Sql dom: compile time checking of dynamic sql statements. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 88–96, New York, NY, USA, 2005. ACM Press.

[28] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 74–89, New York, NY, USA, 1997. ACM.

[29] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005.

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

[31] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Twentieth IFIP International Information Security Conference (SEC'05)*, pages 295–308, 2005.

[32] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[33] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *International Conference on Software Engineering (ICSE)*, pages 478–488, New York, NY, USA, 2002. ACM Press.

[34] Roger E. Sanders. *ODBC 3.5 Developer's Guide*. McGraw-Hill Professional, 1998.

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

[36] M. Sutton. How prevalent are SQL injection vulnerabilities? Technical report, `http://portal.spidynamics.com/blogs/msutton/archive/2006/09/26/How-Prevalent-Are-SQL-Injection-Vulnerabilities_3F00_.aspx`, September 2006.

[37] Peter Thiemann. Grammar-based analysis of string expressions. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 59–70, New York, NY, USA, 2005. ACM.

[38] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41, New York, NY, USA, 2007.

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

[40] Y. Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Usenix Security Symposium*, pages 179–192, July 2006.