Sound and Precise Analysis of Web Applications for Injection Vulnerabilities [*]

Gary Wassermann Zhendong Su
University of California, Davis
{wassermg, su}@cs.ucdavis.edu

Abstract

Web applications are popular targets of security attacks. One common type of such attacks is SQL injection, where an attacker exploits faulty application code to execute maliciously crafted database queries. Both static and dynamic approaches have been proposed to detect or prevent SQL injections; while dynamic approaches provide protection for deployed software, static approaches can detect potential vulnerabilities before software deployment. Previous static approaches are mostly based on tainted information flow tracking and have at least some of the following limitations: (1) they do not model the precise semantics of input sanitization routines; (2) they require manually written specifications, either for each query or for bug patterns; or (3) they are not fully automated and may require user intervention at various points in the analysis. In this paper, we address these limitations by proposing a *precise*, *sound*, and *fully automated* analysis technique for SQL injection. Our technique avoids the need for specifications by considering as attacks those queries for which user input changes the intended syntactic structure of the generated query. It checks conformance to this policy by conservatively characterizing the values a string variable may assume with a context free grammar, tracking the nonterminals that represent user-modifiable data, and modeling string operations precisely as language transducers. We have implemented the proposed technique for PHP, the most widely-used web scripting language. Our tool successfully discovered previously unknown and sometimes subtle vulnerabilities in real-world programs, has a low false positive rate, and scales to large programs (with approx. 100K loc).

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation

General Terms Languages, Security, Verification

Keywords Static Analysis, String Analysis, Web Applications

[*] This research was supported in part by NSF NeTS-NBD Grant No. 0520320, NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1. Introduction

Web applications enable much of today's online business including banking, shopping, university admissions, and various governmental activities. Anyone with a web browser can access them, and the data they manage typically has significant value both to the users and to the service providers. Consequently, vulnerabilities that allow an attacker to compromise a web application's control of its data pose a significant threat. SQL command injection vulnerabilities (SQLCIVs) comprise most of this class. Not only are SQLCIVs serious, but they are pervasive. In 2006, 14% of the CVEs (*i.e.*, reported vulnerabitilities) were SQLCIVs, making SQL injection the second most frequently reported security threat [9]. Some web security analysts speculate that because web applications are highly accessible and databases often hold valuable information, the percentage of SQL injection attacks being executed is significantly higher than the percentage of reported vulnerabilities would suggest [26].

SQLCIVs are common primarily because applications typically communicate with backend databases by passing queries as strings. Figure 1 shows the typical three-tiered web application architecture and illustrates the communication among the tiers: web browsers provide a ubiquitous user interface, application servers manage the business logic, and back-end databases store the persistent data. Because the application layer uses a low-level, queries-as-strings API to communicate with the database, the application constructs queries via low-level string manipulation and treats untrusted user inputs as isolated lexical entities. This is especially common in web applications written in scripting languages such as PHP, which generally do not provide more sophisticated APIs and use strings as the default representation for data and code. Consequently, some paths in application code may incorporate user input unmodified or unchecked into database queries. The modifications/checks of user input on other paths may not adequately constrain the input to function in the generated query as the application programmer intended (see Figure 2 for an example).

1.1 Existing Approaches

Many approaches have been proposed for preventing SQL injection attacks, both dynamic [7, 23, 24, 25] and static [12, 18, 31]. Runtime approaches are useful for protecting deployed software, but static approaches are desirable during software development and testing for a number of reasons. First, a single programming error often manifests itself as multiple different bugs, so statically verifying code to be free from one kind of error (*e.g.*, static type checking) helps to reduce the risk of other errors. Second, the overhead that general techniques incur significantly exceeds the overhead of appropriate, well-placed checks on untrusted input. Even if the network latency dominates the overhead of a runtime check for a single user, the added overhead can prevent a server from functioning

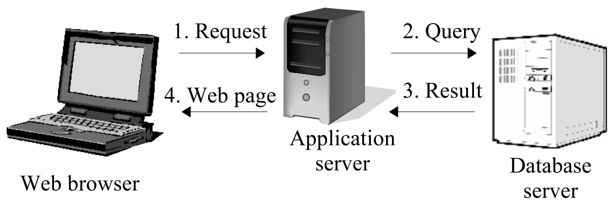

Figure 1. Web application architecture.

effectively under a heavy load of requests. Finally, some runtime techniques [23, 24] require a modified runtime system, which constitutes a practical limitation in terms of deployment and upgrading.

Static analyses to find SQLCIVs have also been proposed, but none of them runs without user intervention and can guarantee the absence of SQLCIVs. String analysis-based techniques [3, 20] use formal languages to characterize conservatively the set of values a string variable may assume at runtime. They do not track the source of string values, so they require a specification, in the form of a regular expression, for each query-generating point or *hotspot* in the program — a tedious and error-prone task that few programmers are willing to do. Static taint analyses [12, 18, 31] track the flow of tainted (*i.e.*, untrusted) values through a program and require that no tainted values flow into hotspots. Because they use a binary classification for data (tainted or untainted), they classify functions as either being sanitizers (*i.e.*, all return values are untainted) or being security irrelevant. Because the policy that these techniques check is context-agnostic, it cannot guarantee the absence of SQLCIVs without being overly conservative. For example, if the `escape_quotes` function (which precedes quotes with an “escaping” character so that they will be interpreted as character literals and not as string delimiters) is considered a sanitizer, an SQLCIV exists but would not be found in an application that constructs a query using escaped input to supply an expected numeric value, which need not be delimited by quotes. Additionally, static taint analyses for PHP typically require user assistance to resolve dynamic includes (a construct in which the name of the included file is generated dynamically).

1.2 Our Approach

We propose a sound, automated static analysis algorithm to overcome the limitations described above. It is grammar-based; we model string values as context free grammars (CFGs) and string operations as language transducers following Minamide [20]. This string analysis-based approach tracks the effects of string operations and retains the structure of the values that flow into hotspots (*i.e.*, where query construction occurs). If all of each string in the language of a nonterminal comes from a source that can be influenced by a user, we label the nonterminal with one of two labels. We assign a “`direct`” label if a user can influence the source directly (as with `GET` parameters) and a “`indirect`” label if a user can influence the source indirectly (as with data returned by a database query). Such labeling tracks the source of string values. We use a syntax-based definition of SQL injection attacks [25], which requires that input from a user be syntactically isolated within a generated query. This policy does not need user-provided specifications. Finally, we check policy conformance by first abstracting the labeled subgrammars out of the generated CFG to find their contexts. We then use regular language containment and context free language derivability [28], to check that each subgrammar derives only syntactically isolated expressions.

We have implemented this analysis for PHP, and applied it to several real-world web applications. Our tool scales to large code bases — it successfully analyzes the largest PHP web application

```
...
01 isset ($_GET['userid']) ?
02     $userid = $_GET['userid'] : $userid = '';
03 if ($USER['groupid'] != 1)
04 {
05     // permission denied
06     unp_msg($gp_permserror);
07         exit;
08 }
09 if ($userid == '')
10 {
11     unp_msg($gp_invalidrequest);
12     exit;
13 }
14 if (!eregi('[0-9]+', $userid))
15 {
16     unp_msg('You entered an invalid user ID.');
17     exit;
18 }
19 $getuser = $DB->query("SELECT * FROM `unp_user`"
20                      ."WHERE userid='$userid'");
21 if (!$DB->is_single_row($getuser))
22 {
23     unp_msg('You entered an invalid user ID.');
24     exit;
25 }
...
```

Figure 2. Example code with an SQLCIV.

previously analyzed in the literature (about 100K loc). It discovered many vulnerabilities, some previously unknown and some based on insufficient filtering, and generated few false positives.

2. Overview

In order to motivate our analysis, we first present the policy that defines SQLCIVs, and then give an overview of how our analysis checks web applications against that policy.

2.1 SQL Command Injection Vulnerabilities

This section illustrates SQLCIVs and formally defines them.

2.1.1 Example Vulnerability

Figure 2 shows a code fragment excerpted from Utopia News Pro, a real-world news management system written in PHP; we will use this code to illustrate the key points of our algorithm. This code authenticates users to perform sensitive operations, such as managing user accounts and editing news sources. Initially, the variable `$userid` gets assigned data from a `GET` parameter, which a user can easily set to arbitrary values. The code then performs two checks on the value of `$userid` before incorporating it into an SQL query. The query should return a single row for a legitimate user, and no rows otherwise. From line `14` it is clear that the programmer intends `$userid` to be numeric, and from line `20` it is clear that the programmer intends that `$userid` evaluate to a single value in the SQL query for comparison to the `userid` column. However, because the regular expression on line `14` lacks anchors (‘`^`’ and ‘`$`’ for the beginning and end of the string, respectively), any value for `$userid` that has at least one numeric character will be included into the generated query. If a user sets the `GET` parameter to “`1'; DROP TABLE unp_user; --`”, this code will send to the database the folloing query:

```
SELECT * FROM `unp_user` WHERE userid='1';
DROP TABLE unp_user; --'
```

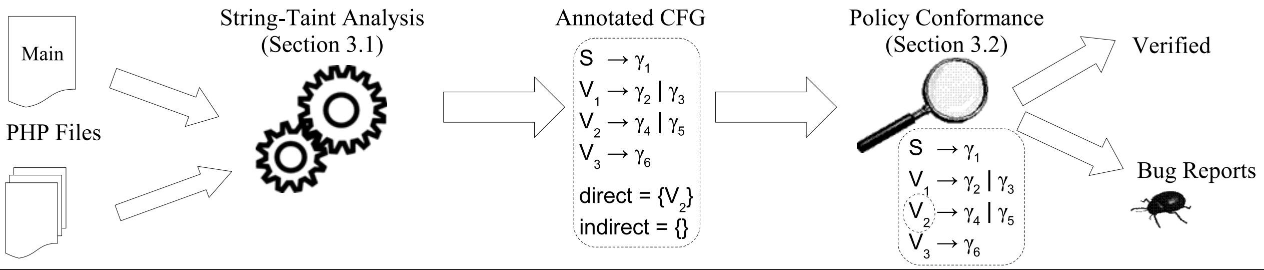

Figure 3. SQLCIV analysis workflow.

and delete user account data.

2.1.2 Definition of SQLCIVs

This section presents the formal definition of command injection attacks that serves as the basis for the policy we seek to enforce.

A web application takes input strings, which it may modify, and generates a query in the form of a string, usually by combining constant strings and filtered inputs. To reflect this, we previously defined a *web application* as follows [25]:

Definition 2.1 (Web Application)**.** A *web application* $P : \langle\Sigma^*, \ldots, \Sigma^*\rangle \rightarrow \Sigma^*$ is a mapping from user inputs (over an alphabet Σ) to query strings (over Σ). In particular, P is given by $\{\langle f_1, \ldots, f_n\rangle, \langle s_1, \ldots, s_m\rangle\}$ where:

- $f_i : \Sigma^* \rightarrow \Sigma^*$ is an *input filter;*
- $s_i : \Sigma^*$ is a *constant string.*

The argument to P is an k-tuple of *input strings* $\langle i_1, \ldots, i_k\rangle$, and P returns a *query* $q = q_1 + \ldots + q_\ell$ where, for $1 \leq j \leq \ell$,

$$q_j = \begin{cases} s & \text{where } s \in \{s_1, \ldots, s_m\} \\ f(i) & \text{where } f \in \{f_1, \ldots, f_n\} \wedge i \in \{i_1, \ldots, i_k\} \end{cases}$$

That is, each q_j is either a static string or a filtered input.

Definition 2.1 primarily serves to help define SQL command injection attacks. This definition does not allow certain string operations that real web applications can do, but, significant for a static analysis, it does allow arbitrary control constructs, arbitrary filtering, and concatenation. Sections 3.1.2 and 3.1.3 address other operations that web applications do and how to handle them with a static analysis.

The syntactic structure of the generated query determines how it will be evaluated. We state here our definition of SQL command injection attacks [25] in terms of sentential forms.

Let $G = (V, \Sigma, S, R)$ be a context-free grammar with nonterminals V, terminals Σ, a start symbol S, and productions R. Let '$\Rightarrow_G$' denote "derives in one step" so that $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ if $A \rightarrow \gamma \in R$, and let '$\Rightarrow_G^*$' denote the reflexive transitive closure of '$\Rightarrow_G$.' If $S \Rightarrow_G^* \gamma$, then γ is a "sentential form." The following definition formalizes syntactic confinement:

Definition 2.2 (Syntactic Confinement)**.** Given a grammar $G = (V, \Sigma, S, R)$ and a string $\sigma = \sigma_1\sigma_2\sigma_3 \in \Sigma^*$, σ_2 is *syntactically confined* in σ iff there exists a sentential form $\sigma_1 X \sigma_3$ such that $X \in V$ and $S \Rightarrow_G^* \sigma_1 X \sigma_3 \Rightarrow_G^* \sigma_1\sigma_2\sigma_3$.

In the attack shown in Section 2.1.1, the user-provided substring is not syntactically confined. The criterion of syntactic confinement effectively distinguishes SQL injection attacks from safe queries [25]. We therefore attempt to enforce the policy that user-provided substrings be syntactically confined.

Definition 2.3 (SQL Command Injection Attack)**.** Given a web application $P = \{\langle f_1, \ldots, f_n\rangle, \langle s_1, \ldots, s_m\rangle\}$ and a query string q constructed from the input $\langle i_1, \ldots, i_k\rangle$, q is a *command injection attack* if there exists $i \in \{i_1, \ldots, i_k\}$ and $f \in \{f_1, \ldots, f_m\}$ such that $q = q_1 + f(i) + q_2$ and $f(i)$ is not syntactically confined in q with respect to the SQL grammar.

A web application has an SQLCIV if it may generate an SQL command injection attack.

2.2 Analysis Overview

Our analysis takes PHP files as input and returns as output either a list of bug reports or the message "verified."

In order to provide useful bug reports, we first categorize sources of untrusted input as being either *direct* or *indirect*. Direct sources, such as `GET` parameters, provide data immediately from users; indirect sources, such as results from a database query, provide data from a source whose data may come from untrusted users. In practice, the rise of attacks from indirect sources is less severe than that of standard injection attacks for two reasons. First, programs often regulate which data is allowed to go into the database (or other sources), and second, attackers must pass through more steps and take more time to execute an indirect attack than to execute a standard injection attack.

Figure 3 shows a high-level overview of our analysis algorithm. It has two main phases. The first phase generates a conservative, annotated approximation of the query strings a program may generate; the annotations show which substrings in the query string are untrusted, *i.e.*, are from either DIRECT or INDIRECT sources. This phase is based on existing string analysis techniques [20] augmented to propagate taint information. The string-taint analyzer takes as input a PHP file that provides the top-level code for a web page (analogous to a `main` function in C). As it encounters dynamic include statements, it determines the possible string values of the argument to the `include`, and analyzes those files as well. The string-taint analyzer represents the set of query strings using an annotated context free grammar (CFG) — the nonterminals whose sub-languages represent untrusted strings are labeled with "`direct`" or "`indirect`," as appropriate. We choose to represent sets of strings with CFGs for several reasons: (1) tainted substring boundaries can be represented simply by labeling certain nonterminals; (2) our policy is grammar-based, and a CFG representation can capture context-free query construction that follows the policy; (3) regular expression-based string operations (common in PHP) can be represented as finite state transducers (FSTs), and the image of a CFG over an FST is context free.

The second phase of our analysis takes the annotated CFG produced by the first phase, and checks whether all strings in the language of the CFG are safe, *i.e.*, they are not SQL command injection attacks according to Definition 2.3. This analysis checks for common cases (of both SQL command injection attacks and attack-free grammars) efficiently by (1) abstracting the subgrammars that represent untrusted substrings out of the larger CFG, (2) determining the syntactic contexts of those subgrammars within the larger

query	$\rightarrow$ *query1* `'`
query1	$\rightarrow$ *query2 userid*
query2	$\rightarrow$ *query3* `WHERE userid='`
query3	$\rightarrow$ `SELECT * FROM `unp_user``
userid	$\rightarrow$ *GETuid*
GETuid	$\rightarrow \Sigma* \text{ [0–9] } \Sigma*$

direct = {*GETuid*} indirect = {}

Figure 4. Grammar productions of possible query strings from Figure 2.

CFG, and (3) checking for (the absence of) policy violating strings in the languages of the subgrammars. For large grammars, this is significantly more efficient than checking the language of the generated CFG as a whole. If the policy conformance checker finds any violations, it issues a bug report. Because this algorithm is sound, if it does not issue any bug reports, the PHP code is guaranteed to be free from SQLCIVs.

To illustrate this algorithm on the example code in Figure 2, the string-taint analysis will produce the grammar productions shown in Figure 4; the annotations are shown in terms of sets of nonterminals annotated with "direct" and "indirect," respectively. The regular expression notation on the right hand side of the last rule is notational shorthand intended to simplify the presentation. The grammar for *userid* reflects the regular expression match on line `14`, because the string-taint analyzer propogates the regular expression predicate. The nonterminal *GETuid* has the label "direct," because it represents strings from a `GET` parameter.

The policy-conformance checker then receives this labeled grammar. The check first replaces the annotated *GETuid* nonterminal with a new terminal $t \notin \Sigma$. By intersecting this modified grammar with an appropriate regular language, the checker finds that for all sentential forms $\sigma_1.GETuid.\sigma_2$ derivable from *query*, *GETuid* is between quotes in the syntactic position of a string literal. The checker therefore uses another regular language intersection to check the language rooted at *GETuid* for un-escaped quotes. When it finds them, it issues a bug report. The checker does not only check for the case of string literals, but that suffices for this example.

3. Analysis Algorithm

This section describes our analysis algorithm in detail.

3.1 String-Taint Analysis

The first phase of our analysis combines ideas from static taint analysis with string analysis.

3.1.1 Adapting String Analysis

String analysis has the goal of producing a representation of all strings values that a variable may assume at a given program point. This goal does not imply any relationship between the structure of that representation and the way that the program produces those values. If the string analysis represents languages as finite automata and it determinizes intermediate results, the final DFA will have little relation to the program's dataflow [3]. Our analysis has the goal of producing not only a representation of all string values that a variable may assume, but also a function from string values to substrings whose values come from direct or indirect sources. In terms of Definition 2.3, we need to identify occurences of $f(i)$ in each query string q.

Section 2.2 gives as one reason for using CFGs to represent sets of query strings that tainted substring boundaries can be represented by labeling certain nonterminals—thus the strings' deriva-

(a)
```
$X = $UNTRUSTED;
if ($A) {
   $X = $X."s";
} else {
   $X = $X."s";
}
$Z = $X;
```

(b)
```
$X1 = $UNTRUSTED;
if ($A) {
   $X2 = $X1."s";
} else {
   $X3 = $X1."s";
}
$X4 = φ($X2, $X3);
$Z = $X4;
```

(c)
$$\begin{array}{ll} UNTRUSTED & \rightarrow \Sigma^* \\ X_1 & \rightarrow UNTRUSTED \\ X_2 & \rightarrow X_1\texttt{s} \\ X_3 & \rightarrow X_1\texttt{s} \\ X_4 & \rightarrow X_2 \mid X_3 \\ Z & \rightarrow X_4 \end{array}$$

Figure 5. Grammar reflects dataflow.

tions provide the function from strings to tainted substrings. In addition to that reason, a natural way to design and implement a CFG-based string analysis produces CFGs that reflect the program's dataflow, so taint annotations applied at untrusted sources appear in the final CFG. Minamide designed his string analysis this way, so we review the main steps of his analysis here and show how to adapt it to track taint information [20].

The contrived example program in Figure 5a serves to show that the generated grammar reflects the program's dataflow. The first step of the string analysis translates the program into static single assignment form, as shown in Figure 5b. SSA form makes data-dependencies explicit, so translating each assignment statement into a grammar production yields a CFG that reflects the program's dataflow, as in Figure 5c. By simply annotating the nonterminals corresponding to direct and indirect sources appropriately, we have a string-taint anlysis for programs with concatenation, assignments, and control flow.

In general, right hand sides of assignment statements may contain string functions, such as `escape_quotes()`, which adds a slash before each quote character in its argument. Translating assignments into grammar productions then yeilds an extended CFG that has functions in its productions' right hand sides. Converting extended CFGs into standard CFGs requires some approximation, and Minamide models string operations as finite state transducers (FSTs) in order to capture their effects and make the approximation reasonably precise. Section 3.1.2 describes how FSTs can model string operations and how we track annotated sources through them in more detail.

3.1.2 Tracking Substrings through Filters

Definition 2.1 specifies that web applications can apply functions on strings to untrusted inputs. In order to avoid reporting many false positives, the string-taint analyzer must model the effects of filters and propagate annotations through them. This section reviews how Minamide's string analysis models the effects of filters and then describes how we adapt these techniques.

A transducer is an automaton with output. A finite state transducer is similar to a Mealy machine, except that a finite state transducer has one or more final states and may be non-deterministic. Many string operations that PHP provides as library functions behave as finite state transducers. For example, `str_replace` takes three strings as arguments: a pattern, a replacement, and a subject. The FST in Figure 6 describes the effects of `str_replace` when the pattern is '`''`' and the replacement is '`'`.' The notation 'c_1`/`c_2' on the transitions means that on input character c_1, the transition can be taken and it will output c_2. In Figure 6, A matches any character except '`'`.' The string analysis converts a grammar production

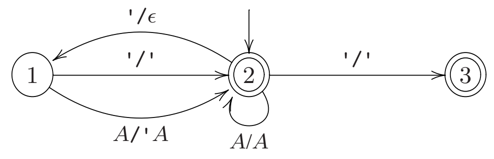

Figure 6. A finite state transducer equivalent of the function `str_replace("''", "'", $B)`; $A \in \Sigma \setminus \{\texttt{'}\}$.

with a string operation, such as

$$x \rightarrow \texttt{escape\_quotes}(y)$$

into a standard grammar production by finding the image of the CFG rooted at the operation's argument (y) over the FST that the string operations represents.

A CFG has a *cycle* if there exists a sentential form derivable from a nonterminal that contains the nonterminals. If an extended grammar has the production shown above, and if

$$y \Rightarrow_G^* \alpha x \beta$$

then the `escape_quotes` operation occurs in a cycle. String operations that occur in cycles within the extended CFG must be approximated because the complete CFG rooted at the operation's argument (y) cannot be constructed independently of the string operation.

Some string operations are more expressive than finite state, or even context free, transducers. For example, PHP provides a regular expression-based replace function, `preg_replace`. Its three arguments are: a regular expression pattern, a parameterized replacement, and a subject. Within the replacement, an occurence of "\\n," where n is a number, represents the string matched by the expression between the n^{th} open parenthesis and its matching close parenthesis in the pattern. As an example,

```
preg_replace("/a([0-9]*)b/",
             "x\\1\\1y",
             "a01ba234b") = "x0101yx234234y"
```

The "`\\1`" puts the substring matched by the expression within the first pair of parentheses (because the number is `1`) into the output. Although the image of a CFL under a regular expression replacement is not necessarily context free (because of the ability to insert multiple copies of a regular expression match, as above), Mohri and Sproat describe how to approximate it using two FSTs [21].

The string analysis also uses a similar technique to maintain precision from conditional expressions when constructing the extended CFG. If the condition is a regular expression match, as on line `14` in Figure 2, the string analysis adds an intersection with the condition's regular expression to the beginning of the `then` branch and an intersection with the complement of the regular expression to the `else` branch.

An adaptation of the standard context free language-reachability algorithm [19] computes the intersection of a CFG and an FSA as a CFG without constructing an intermediate push-down automaton, and we add to the algorithm to propagate annotations. Figure 7 shows the algorithm with our additions: the function TAINTIF() and the two calls to it on lines `20` and `28` of INTERSECT(). The following theorem states that this algorithm propagates annotations appropriately.

Theorem 3.1. Given C' = INTERSECT(C, F), s $\in \mathcal{L}(C) \cap \mathcal{L}(F)$, and a parse tree p of s under C, there exist s_1, s_2, and s_3 such that $s_1 s_2 s_3 = s$ and s_2 is derivable from a direct-labeled nonterminal in p iff there exists a parse tree p' of s under C' such that s_2 is derivable from a direct-labeled nonterminal in p'.

```
INTERSECT($G = \langle V_0, \Sigma_0, S_0, R_0 \rangle$, $FSA = \langle Q, \Sigma, \delta, q_0, q_f \rangle$)
 1  $\langle V, \Sigma, S, R \rangle \leftarrow$ NORMALIZE($\langle V_0, \Sigma_0, S_0, R_0 \rangle$)
 2  $V' \leftarrow \emptyset$
 3  $R' \leftarrow \emptyset$
 4  for each $(q_i, \sigma, q_j)$ in $\delta$
 5  do $V' \leftarrow V' \cup \{\sigma_{ij}\}$
 6      $R' \leftarrow R' \cup \{\sigma_{ij} \rightarrow \sigma\}$
 7  /* $|rhs| = 0$ */
 8  for each $X \rightarrow \epsilon$ in $R$
 9  do for each $q_i$ in $Q$
10      do $V' \leftarrow V' \cup \{X_{ii}\}$
11          $R' \leftarrow R' \cup \{X_{ii} \rightarrow \epsilon\}$
12  $WkLst \leftarrow V'$
13  for each $\alpha_{ij}$ in $WkLst$
14  do $WkLst \leftarrow WkLst \setminus \{\alpha_{ij}\}$
15      /* $|rhs| = 1$ */
16      for each $X \rightarrow \alpha$ in $R$
17      do if not ($X_{ij}$ in $V'$)
18             then $WkLst \leftarrow WkLst \cup \{X_{ij}\}$
19                  $V' \leftarrow V' \cup \{X_{ij}\}$
20                  TAINTIF($X, X_{ij}$)
21          $R' \leftarrow R' \cup \{X_{ij} \rightarrow \alpha_{ij}\}$
22      /* $|rhs| = 2$ */
23      for each $X \rightarrow \alpha\beta$ in $R$
24      do for each $\beta_{jk}$ in $V'$
25          do if not ($X_{ik}$ in $V'$)
26                 then $WkLst \leftarrow WkLst \cup \{X_{ik}\}$
27                      $V' \leftarrow V' \cup \{X_{ik}\}$
28                      TAINTIF($X, X_{ik}$)
29              $R' \leftarrow R' \cup \{X_{ik} \rightarrow \alpha_{ij}\beta_{jk}\}$
30  if $S_{0f}$ in $V'$
31     then return $\langle V', \Sigma, S_{0f}, R' \rangle$
32     else return $\langle \{S'\}, \{\}, S', \{\} \rangle$

NORMALIZE($G = \langle V, \Sigma, S, R \rangle$)
 1  $R' \leftarrow \emptyset$
 2  $WkLst \leftarrow R$
 3  for each $X \rightarrow [\gamma]$ in $WkLst$
 4  do $WkLst \leftarrow WkLst \setminus \{X \rightarrow [\gamma]\}$
 5      if $length[\gamma] > 2$
 6         then $X' \leftarrow$ FRESHVAR()
 7              $V \leftarrow V \cup \{X'\}$
 8              $R' \leftarrow R' \cup \{X \rightarrow head[\gamma]X'\}$
 9              $WkLst \leftarrow WkLst \cup \{X' \rightarrow tail[\gamma]\}$
10         else $R' \leftarrow R' \cup \{X \rightarrow [\gamma]\}$
11  return $\langle V, \Sigma, S, R' \rangle$

TAINTIF($X_1, X_2$)
1  if HASLABEL($X_1$, direct)
2     then ADDLABEL($X_2$, direct)
3  if HASLABEL($X_1$, indirect)
4     then ADDLABEL($X_2$, indirect)
```

Figure 7. Taint propagation in CFG-FSA intersection.

The proof is by a straightforward induction on the height of the derivation of s. Due to space constraints, we omit the proof here. The case for "INDIRECT" is identical.

The algorithm for finding the image of a CFG over an FST is similar to the CFG-FSA intersection algorithm, except that the FST's output symbols replace the CFG's terminals as they match

$$
\begin{array}{lcl}
\texttt{explode}(s_1, s_2) & = & \texttt{expld}(s_1, s_2, \texttt{[ ]}, \epsilon) \\
\texttt{expld}(s_1, s_1, L, s) & = & L@\texttt{[}s\texttt{]} \\
\texttt{expld}(s_1, s_2, L, s), |s_2| \leq |s_1| & = & L@\texttt{[}s\hat{\ }s_2\texttt{]} \\
\texttt{expld}(s_1, s_1\hat{\ }s_2, L, s) & = & \texttt{expld}(s_1, s_2, L@\texttt{[}s\texttt{]}, \epsilon) \\
\texttt{expld}(s_1, c\hat{\ }s_2, L, s), |c| = 1 & = & \texttt{expld}(s_1, s_2, L, s\hat{\ }c)
\end{array}
$$

Figure 8. Semantics of `explode`.

the FST's input symbols. The modifications for propogating taint information are the same for that algorithm as in Figure 7, and the proof of correctness is analogous to the proof of Theorem 3.1.

Definition 2.1 only allows string concatenation after input filters. The taint-propagating algorithm in Figure 7 correctly propagates tainted substring boundaries even for filters that operate on inputs concatenated with other strings. Thus we extend the definition of SQLCIVs to web applications with operations beyond those that Definition 2.1 allows and still check for SQLCIVs with high precision.

3.1.3 Handling Other String Operations

Real-world web applications also perform operations involving strings that do not simply map strings to strings. For each such operation, we must determine how substrings in the input map to substrings in the output and propagate annotations accordingly. We use as a straightforward but representative example the `explode` function, which takes two string arguments: a delimiter and a subject. It returns an array of substrings formed by splitting the subject on boundaries formed by the delimiter. Figure 8 shows the semantics of `explode`. Because the strings that it returns are taken directly from the subject, the meaning of untrusted substring flow is clear.

The string analysis models the effects of `explode` accurately, except that it loses the order of the strings in the returned array—it produces a grammar whose language is that set of strings. The algorithm (due to Minamide [20]) uses two FSTs constructed from the delimiter, and because we propagate labels through FSTs correctly (see Section 3.1.2), we track tainted substrings accurately through the `explode` function. Space limitations prevent us from giving a full presentation of the algorithm here.

3.2 Policy-Conformance Analysis

The second phase of our analysis checks the generated, annotated grammar for SQL injection attacks. In most cases, programmers intend that inputs take the syntactic position of literals. Section 3.2.1 describes our checks for this case, and Section 3.2.2 presents our approach for the case when the input may be derived from an arbitrary nonterminal in the reference (SQL) grammar.

3.2.1 Untrusted Substrings as Literals

This section describes how we attempt for each annotated nonterminal X either to verify that all strings derivable from X are syntactically confined or to find that some string derivable from X are not syntactically confined. We apply the algorithm described in Section 3.2.2 to nonterminals for which the checks in this section fail to provide conclusive results.

The first check attempts to find untrusted substrings that cannot be syntactically confined in any SQL query. In particular, because quotes delimit string literals in SQL, if any untrusted substring has an odd number of un-escaped quotes (escaped quotes represent characters rather than delimiters in string literals), it cannot be syntactically confined. The grammar generated by the string-taint analysis reflects the program's dataflow, so the strings derivable from labeled nonterminals are the possible untrusted substrings in generated SQL queries. Let V_l be the set of labeled nonterminals in V. For each $X \in V_l$, if

$\emptyset \neq \mathcal{L}(V, \Sigma, X, R) \cap \mathcal{L}($
```
/^(([^']|\')*[^\])?'
 ((([^']|\')*[^\])?'
  (([^']|\')*[^\])?')*
 ([^']|\')*$/ )
```

then there exists a string derivable from X that is not syntactically confined (the Perl regular expression matches strings with an odd number of unescaped quotes), and we remove X from V_l.

The second check finds the nonterminals in V_l that occur only in the syntactic position of string literals and, for each one, either verifies it as safe or finds that it derives some unconfined string. The algorithm identifies the syntactic position of labeled nonterminals by creating from the grammar production set R a new production set R_t: for each labeled nonterminal $X \in V$, replace right-hand-side occurences of X in R with a fresh terminal $t_X \notin \Sigma$ and add t_X to Σ. For each labeled $X \in V$, if for all strings $\sigma_1 t_X \sigma_2 \in \mathcal{L}(V, \Sigma, S, R_t)$, σ_1 has an odd number of unescaped quotes, then X only occurs in the syntactic position of a string literal. The following implements this check:

$\emptyset = \mathcal{L}(V, \Sigma, S, R') \cap \mathcal{L}($
```
/^[^']*
 ('(([^']|\')*
    (([^\][\\]+)|[^'\]))?
  '[^']*)*
 tX.*$/ )
```

For each $X \in V_l$ for which the test above succeeds, if any $\sigma \in \mathcal{L}(V, \Sigma, X, R)$ has unescaped quotes in it, X derives unconfined strings; otherwise X is safe. We then remove X from V_l.

The third check attempts to identify those remaining nonterminals in V_l that only derive numeric literals. For each $X \in V_l$, if

$\emptyset = \mathcal{L}(V, \Sigma, X, R) \cap \mathcal{L}($
```
/^(([^0-9.+-].*[^0-9.]/)
 |([.].*[.])) )
```

then X derives only numeric literals and is safe; remove X from V_l.

Finally, if X can produce a non-numeric string outside of quotes, it likely represents an SQLCIV. To confirm this, we check whether X can derive any of a given set of strings that cannot be syntactically confined (*e.g.*"`DROP WHERE`," "`--`," etc.). If it can, then X is unsafe, and we remove it from V_l.

3.2.2 Untrusted Substrings Confined Arbitrarily

If any nonterminals remain in V_l after the checks in Section 3.2.1, we wish to check whether each string derivable from them is derivable from some nonterminal in the SQL grammar. In general, context free language inclusion is undecidable, but we can approximate it by checking grammar derivability, *i.e.*, whether the generated grammar is derivable from the SQL grammar [28].

Definition 3.2 (Derivability)**.** Grammar $G_1 = (V_1, \Sigma, S_1, R_1)$ is *derivable* from grammar $G_2 = (V_2, \Sigma, S_2, R_2)$ iff

$$
\begin{array}{l}
\exists\, \Phi : (V_1 \cup \Sigma) \to (V_2 \cup \Sigma) \\
\quad \Phi(S_1) = S_2 \wedge \\
\quad \forall\, s \in \Sigma \;\; \Phi(s) = s \wedge \\
\quad\quad \forall\, (X \to \gamma) \in R_1 \;\; \Phi(X) \Rightarrow^*_{G_2} \Phi^*(\gamma)
\end{array}
$$

where Φ^* is Φ lifted to $(V_1 \cup \Sigma)^*$, *i.e.*,

$$
\begin{array}{rcl}
\Phi^*(\epsilon) & = & \epsilon \\
\Phi^*(\alpha) & = & \Phi(\alpha) \quad \text{for } \alpha \in V_1 \cup \Sigma \\
\Phi^*(\alpha\beta) & = & \Phi^*(\alpha)\Phi^*(\beta)
\end{array}
$$

Lemma 3.3. *If G_1 is derivable from G_2, then $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$.*

We check derivability using an extension of Earley's parsing algorithm [4] that parses sentential forms and treats nonterminals in

G_1 as variables that range over terminals and nonterminals. This algorithm is inspired by and is similar to Thiemann's algorithm [28]. We do not require that the entire generated grammar be derivable from the SQL grammar; we require derivability for the subgrammar rooted at X and all sentential forms that include X. If the derivability check fails, we consider X to be unsafe.

3.3 Soundness

We state and sketch the proof of a soundness result here.

Theorem 3.4 (Soundness)**.** If our analysis algorithm does not report any SQLCIVs for a given web application P, then P has no SQLCIVs.

Proof. The string analysis produces a CFG G from web application P that derives all strings that P may generate as query strings [20]. The algorithm for constructing G reflects P's dataflow so that for assignments and concatenation, labels on nonterminals from untrusted sources accurately identify untrusted substrings. By Theorem 3.1, the CFGs constructed as the intersection of a CFG and an FSA, or the image of a CFG over an FST, is labeled to reflect the boundaries of untrusted substrings. The conformance checking algorithm from Section 3.2 generates an error message on each labeled nonterminal unless the algorithm can verify it to derive only syntactically confined strings, as required by Definition 2.3. $\square$

4. Implementation

We implemented our technique for PHP, using and modifying Minamide's string analyzer. In addition to the changes described previously (adding information flow tracking and checks on the generated grammars), we made the analyzer more automated in two ways. First, we added specifications for 243 PHP functions. Second, we enhanced its support for dynamic includes. Previously, the analyzer would fail if it reached an include statement, and the grammar it had generated for the include statement's argument had an infinite language. For example, if the analyzer recorded the possible values for `$choice` as being Σ^*, the analyzer would fail at:

```
include("e107_languages/lan_".$choice.".php");
```

We address this by considering the file and directory layout to be part of the specification. If the analyzer encounters such an include statement, it builds a regular expression representation of the directory layout starting from the analyzed project's root. It then intersects the (finite) language of this regular expression with the language of the grammar to find the list of files to include. This language-based approach does not model the full semantics of paths (*e.g.*, "`..`" as parent directory), but we believe this choice to be appropriate for two reasons. First, we have not encountered cases where the programmer-intended values of variables like `$choice` include "`..`"; and second, security exploits on dynamic inclusion vulnerabilities generally reveal sensitive information stored in files and do not facilitate SQL command injection attacks.

The string analyzer does not support all features for PHP. For example, it includes only limited support for references. We plan to add support for these features, but until full-support is available, we manually approximate unsupported lines of PHP code and verify that the changes do not remove potential errors.

5. Evaluation

This section presents the setup and results of our evaluation.

5.1 Test Subjects

We evaluated our tool on five real-world PHP web applications in order to test its scalability and its false positive rate, and to see what kinds of errors it would find and what would cause false positives. We use the following subjects in our evaluation: e107 and Warp Content Management System are content management systems; EVE Activity Tracker is an activity tracker for integration into existing IGB homepages; and Tiger PHP News System and Utopia News Pro are news management systems. Table 1 lists the size of each of these web applications in terms of the number of files and the number of lines of PHP code. The test suite for another PHP analysis tool [31] includes an earlier version of e107, and we do not know of any database-backed PHP web application with more lines of code. We ran the analysis on a machine with a 3GHz processor and 8GB of RAM running Linux – Fedora Core 5.

```
isset($_GET['newsid']) ?
    $getnewsid = $_GET['newsid'] :
    $getnewsid = false;
if (($getnewsid != false) &&
    (!preg_match('/^[\d]+$/', $getnewsid)))
{
    unp_msg('You entered an invalid news ID.');
    exit;
}
...
if (!$showall && $getnewsid)
{
   $getnews = $DB->query("SELECT * FROM `unp_news`"
                 ."WHERE `newsid`='$getnewsid'"
                 ."ORDER BY `date`DESC LIMIT 1");
}
```

Figure 9. Source of a false positive.

5.2 Accuracy and Bug Reports

The code in Figure 2 shows a vulnerability that our tool found by modeling regular expressions precisely. Two others in Utopia News Pro are similar to this one. Although some of the SQLCIVs that our tool found were trivial, others crossed file and class boundaries. For example, the SQLCIV in e107 comes from a field read from a cookie, which a user can modify, that is used in a query in a different file.

Across this test suite, our tool had a $(5/(19+5)) = 20.8\%$ false positive rate. This false positive rate demonstrate that our approach is effective for finding SQLCIVs and verifying the absence of them.

Our tool produced the false positives that it did because of it does not track information with sufficient precision through type conversions. Figure 9 shows one of the two false positives from Utopia News Pro (the other is similar). The PHP runtime system will dynamically cast between any of the scalar types without complaint. It casts a value of type string to a value of type boolean producing a value of `false` if the string is "" (empty) or "0," and `true` otherwise. To avoid the false positive shown in Figure 9, the analyzer would have to model this conversion in the first conditional expression and propagate its implications beyond the "then" branch. The three false positives from Tiger PHP News System resulted from a hand-written string sanitizing routine. Depending on a character's ASCII value, this routine will either encode it or keep it as is. The string analyzer does not have a map from characters to their ASCII values, so it failed to track the precise effects of this routine. Both of these types of false positives could be avoided by equipping the string analyzer with more information about type conversions.

Evaluating whether indirect error reports represent real errors is difficult because it requires making assumptions about what data can flow into the source (*e.g.*, the database). However, Figure 10 shows one example of an indirect error report that seems to repre-

Name (version)	Files	Lines	Grammar		Time (h:m:s)		Errors		
			Size		String Analysis	SQLCIV Check	direct		indirect
			$\lvert V\rvert$	$\lvert R\rvert$			Real	False	
e107 (0.7.5)	741	132,850	62,350	377,348	3:39:26.23	35:36.12	1	0	4
EVE Activity Tracker (1.0)	8	905	57	1628	0.40	0.06	4	0	1
Tiger PHP News System (1.0 beta 39)	16	7,961	82,082	1,078,768	3:14:06.95	5.39	0	3	2
Utopia News Pro (1.3.0)	25	5,611	5,222	336,362	25:00.08	2:08.69	14	2	12
Warp Content MS (1.2.1)	42	23,003	1,025	73,543	21.10	0.08	0	0	0
Totals							19	5	17

Table 1. Evaluation results.

```
$newsposter = $USER['username'];
$newsposterid = $USER['userid'];
// Verification
if (unp_isEmpty($subject) || unp_isEmpty($news))
{
    unp_msg($gp_allfields);
    exit;
}
if (!preg_match('/^[\d]+$/', $newsposterid))
{
    unp_msg($gp_invalidrequest);
    exit;
}
$submitnews = $DB->query("INSERT INTO `unp_news`"
    ."(`date`, `subject`, `news`, `posterid`,"
        ."`poster`)"
    ." VALUES "
    ."('$posttime','$subject','$news',"
        ."'$newsposterid','$newsposter')");
```

Figure 10. Source of an indirect error report.

sent a true vulnerability. Both `$newsposter` and `$newsposterid` are assigned from the `$USER` array, which is populated elsewhere from the results of a database query. The fact that `$newsposterid` is checked and not `$newsposter` seems to indicate the possibility of unexpected values, and at the least it represents inconsistent programming.

5.3 Scalability and Performance

As stated in Section 4, our string analyzer currently has some limitations in terms of the PHP constructs it supports. Nevertheless, on three of the subjects in our test suite (EVE Activity Tracker, Utopia News Pro, and Warp Content Management System) the analyzer ran successfully. The others include certain currently unsupported constructs, and we manually modified the code to allow the analyzer to continue but without causing any potential errors to be missed. The unsupported construct that we encountered most frequently was the `str_replace` function with array-type arguments, which were generally given statically. We expanded these `str_replace` statements into sequences of `str_replace` statements, each with scalar arguments. These unsupported constructs do not represent a shortcoming in our technique, but only a current limitation in our prototype.

Regarding scalability, we note first that our tool successfully analyzed all of the web applications in our test suite. Table 1 lists the size of the grammars representing SQL queries that our tool generates in terms of the number of nonterminals ($|V|$) and the number of production rules ($|R|$). Next to the grammar size, it lists the time spent on string analysis and the time spent checking the generated grammars for SQLCIVs. Analyzing web applications is different in one key respect. Unlike for many program analysis settings where a code base has a single top-level function that can be passed to an analyzer, each file that represents one page in a web application defines a top level function. In many web applications, most files defining top-level functions include and use the same helper functions in other files, and our tool re-analyzes these included files each time. In such cases, our tool analyzes most of the code in a small fraction of the time required to analyze the whole web application. This also illustrates that straightforward use of memorization or concurrent executions of the analyzer could improve the performance dramatically in some cases.

A few points are particularly noteworthy here. First, the grammar size is not necessarily proportional to the web application size. The query grammar generated from Tiger PHP News System is significantly larger than that of e107, which is over an order of magnitude larger in terms of lines of code. This reflects in some sense the size of the web application devoted to database queries.

Second, the string analysis time is not necessarily proportional to the grammar size. The grammar size reported is only for the grammar representing possible database queries. The string analysis works "eagerly," analyzing some string expressions that have no influence on the generated database queries. This eager analysis introduce significant unnecessary overhead in web applications that process user input for marked up display, such as in an online bulletin board or forum. Tiger PHP News System includes such code, that substitutes html tags for forum equivalents (*e.g.*, `<bold>` for `[bold]`) and designated character sequences for "emoticon" links. Tiger PHP News System is designed to be secure, and it includes a forum with such code. Each regular expression or string replacement function (potentially) causes its argument's grammar to increase by some factor, so that a sequence of these replacement expressions leads to a blow up that is exponential in the number of replacements. We removed two sections of such code from Tiger PHP News System in order to speed up the analysis, but in principle the analyzer could use a backward dataflow analysis to determine which variables may influence a database query, and refrain from analyzing the rest. We expect that this would speed up the analysis significantly. Additionally, dynamic file inclusions can lead to a combinatorial blow up. Each time a file is included, it is inserted *in situ* and its top-level scope is merged into the scope where it is included. If one file has an include statement whose argument is entirely unspecified statically, the analyzer will try to include every other file in the project and with each of them, which ever files they may include. In the case of e107 with 741 files, we had to provide file names for two include statements.

Finally, the SQLCIV checking phase is relatively efficient. Although the grammars had more than one million production rules in some cases, SQLCIV checking never took more than a few minutes, and usually took less.

6. Related Work

In this section we survey closely related work.

6.1 Static String Analysis

The study of static string analysis grew out of the study of text processing programs. An early work to use formal languages (*viz.* regular languages) to represent string values is XDuce [10], a language designed for XML transformations. Tabuchi *et al.* designed regular expression types for strings in a functional language with a type system that could handle certain programming constructs with greater precision than had been done before [27].

Christensen *et al.* introduced the study of static string analysis for imperative (and real-world) languages by showing the usefulness of string analysis for analyzing reflective code in Java programs and checking for errors in dynamically generated SQL queries [3]. They designed an analysis for Java that has FSAs as its target language representation; they chose FSAs because FSAs are closed under the standard language operations. They also applied techniques from computational linguistics to generate good FSA approximations of CFGs [21]. Their analysis, however, does not track the sourced of data, and because it must determinize the FSAs between each operation, it is less efficient than other string analyses and not practical for finding SQLCIVs. Gould *et al.* used this analysis to type check dynamically generated queries, but made approximations that would cause them to miss SQLCIVs [6].

Minamide borrowed techniques from Christensen *et al.* to design a string analysis for PHP that does not approximate CFGs to FSAs, so it can be more efficient and more accurate [20]. He also utilized techniques from computational linguistics (*viz.* language transducers) [22] to improve the precision of his analysis and model the effects of string operations, which are used frequently in scripting languages. His analysis does not track the source of data explicitly, and it is designed to validate dynamically generated HTML, which has a flatter grammar than SQL. For both Minamide and Christensen *et al.*'s analyses, the user must provide regular expression specifications of the permitted queries at each query location. We avoid the need for manually written specifications first by using a general policy based on both dataflow and string structure, and second by adding explicit dataflow information to the grammar's nonterminals in Minamide's analysis.

6.2 Static Taint Checking

Static taint checking is essentially information flow analysis specialized to determine whether data from an untrusted source flows into a sensitive sink. Static taint checking has a long history, but Huang *et al.* were perhaps the first to apply it to SQLCIVs [11]. They used a CQual-like [5] type system to propagate taint information through PHP programs. Livshits and Lam [17] used a precise points-to analysis for Java [30] and queries specified in PQL [16] to find paths in Java programs that allow "raw" input to flow into SQL queries. Both of these tools are sound with respect to the policy they enforce and the language features they support, and both find many vulnerabilities, but both consider all values returned from designated filtering functions to be safe. Because the policy they use says nothing about the context of the user input and the structure of the query, both techniques may miss real SQLCIVs. Additionally, Huang *et al.*'s type system does not support some of PHP's more dynamic features, in part because it does not track string values at all and supporting these features would result in too many false positives.

Jovanovic *et al.* sought to address this last shortcoming with Pixy [12, 13], a static taint analysis for PHP that propagates limited string information and implements a finely tuned alias analysis. Xie and Aiken designed a more precise and scalable analysis for finding SQLCIVs in PHP by using block- and function-summaries [31]. The precision they gained comes at the expense of automation — the user must provide the filenames when the analysis encounters a dynamic include statement, and the user must tell the analysis whether each regular expression encountered in a filtering function is "safe." We are able to make a stronger guarantee about the absence of SQLCIVs because we analyze the possible values of the strings and check conformance to a policy that takes into account the query's structure.

6.3 Runtime Enforcement

Because more information about data and program execution is available at runtime, several groups have proposed techniques to enforce more expressive policies than simply tracking the flow of tainted input, which Perl's taint mode already provides [29]. AMNESIA, by Halfond and Orso, uses Christensen *et al.*'s Java string analyzer to construct a policy requiring user inputs to be single tokens in constructed queries, and enforces that policy at runtime [8]. The effectiveness of this approach is limited by the string analysis' precision. Buehrer *et al.* also enforce a policy that user input must be a single token in the query, but they do not rely on a static analysis [2]. They bound user input, parse the query, and check whether the parse tree retains the same structure when the user input is replaced by a single dummy node. Java provides a `PreparedStatement` API, which forces inputs in queries built with it to be string or numeric literals. Boyd and Keromytis sought to enforce this via instruction set randomization [14], *i.e.*, by randomizing the SQL keywords in the web application, so that users could not guess the keywords [1]. This technique cannot provide guarantees, because the user may guess the randomization key.

Both Nguyen-Tuong *et al.* [23] and Pietraszek and Berghe [24] propose to enforce the same policy for PHP more rigorously. They modify the PHP interpreter to track taint information at the character level, tokenize the completed query, and check whether any tainted characters appear in any tainted characters. A modified interpreter has the advantage that it can add security guarantees to arbitrary web applications, but practical issues of deployment and system maintenance limit such a technique's effectiveness. Xu *et al.* propose a source-to-source translator for C that adds taint tracking [32]. It can be used to ameliorate the system maintenance problem by adding taint-tracking to new versions of the PHP interpreter's source code.

WASP, by Halfond *et al.*, enforces approximately the same policy for Java, but they use *positive tainting*, *i.e.*, they taint trusted strings, and allow only tainted characters in keywords unless the programmer specifies with a regular expression that user input may include certain keywords [7]. Additionally, instead of modifying the JVM, they provide a byte code instrumenter. Su and Wassermann use delimiters to track user input into generated queries, and parse the queries based on a modified grammar to check whether the user input is parsable under any of a permitted set of nonterminals within the query [25]. The policy we enforce allows user input to be parsable under any nonterminal, but in principle we could limit the allowable nonterminals. Although these runtime techniques to prevent SQL injection attacks are more precise than static analyses in general, some SQLCIVs are indicative of larger programming errors. Static analysis can help to find and fix such errors prior to deployment. Additionally, general runtime enforcement techniques incur more runtime overhead than appropriate, well-placed filters, which static analysis can check.

7. Conclusion

In this paper we have proposed a new static analysis algorithm to find SQLCIVs. It characterizes the sets of possible database queries that a web application may generate using context free grammars, and tracks information flow from untrusted sources into those grammars. By using a general definition of SQLCIVs based on the context of untrusted substrings, we avoid the need for manually written policies. Our implementation worked well under evaluation. It was precise, detected unknown vulnerabilities in real-world web applications with few false positives, demonstrating the effectiveness of our approach.

We plan to make three improvements to our tool: first, we plan to extend it to support all of PHP's features; second, we plan to add a backward dataflow analysis to prevent it from analyzing complex string expressions that do not influence database queries; third, we plan to track line numbers from PHP source files through to the grammars' nonterminals in order to improve the quality of the bug reports. We would like to apply the same technique to detecting vulnerabilities that allow cross-site scripting attacks, in which a server may deliver untrusted JavaScript code to be executed by a client browser with the full permissions of the trusted server. We are also interested in integrating our analysis into a broader business logic analysis of web applications [15] in order to track session variables as they flow from one page to another and provide more precise and informative warnings.

Acknowledgments

We thank the PLDI anonymous reviewers for their useful and detailed comments, which helped improve the presentation of this work. We also thank Yasuhiko Minamide for developing the PHP string analyzer that we used in this work and for answering our questions about his tool.

References

[1] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS), LNCS*, volume 2, 2004.

[2] G. T. Buehrer, B. W. Weide, and P. A. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC*, Sept. 2005.

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

[4] J. Earley. An efficient context-free parsing algorithm. *Communications of the Association for Compution Machinery*, 13(2):94–102, 1970.

[5] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[6] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 645–654, May 2004.

[7] W. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, Portland, Oregon, November 2006.

[8] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2005.

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

[10] H. Hosoya and B. C. Pierce. Xduce: A typed xml processing language (preliminary report). In *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*, pages 226–244, London, UK, 2001. Springer-Verlag.

[11] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM Press.

[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[13] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for syntactic detection of web application vulnerabilities. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Ottowa, Canada, June 2006.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. CCS'03*, pages 272–280, 2003.

[15] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proceedings of the 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006. Full version available as BRICS RS-06-10.

[16] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, June 2005.

[17] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[18] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005.

[19] D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 74–89, 1997.

[20] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW'05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

[21] M. Mohri and M. Nederhof. Regular approximation of context-free grammars through transformation. *Robustness in Language and Speech Technology*, pages 153–163, 2001.

[22] M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *Meeting of the Association for Computational Linguistics*, pages 231–238, 1996.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Twentieth IFIP International Information Security Conference (SEC'05)*, 2005.

[24] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[25] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages*, pages 372–382, Charleston, SC, Jan. 2006. ACM Press New York, NY, USA.

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

[27] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language (extended abstract). In *Proceedings of TIP'02 Workshop on Types in Programming*, pages 1–18, July 2002.

[28] P. Thiemann. Grammar-based analysis of string expressions. In *2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 59–70, 2005.

[29] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl (3rd Edition)*. O'Reilly, 2000.

[30] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

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

[32] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Aug. 2006.