Static Detection of Security Vulnerabilities in Scripting Languages

Yichen Xie *Alex Aiken*
Computer Science Department
Stanford University
Stanford, CA 94305
{*yxie,aiken*}*@cs.stanford.edu*

Abstract

We present a static analysis algorithm for detecting security vulnerabilities in PHP, a popular server-side scripting language for building web applications. Our analysis employs a novel three-tier architecture to capture information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural level. This architecture enables us to handle dynamic features of scripting languages that have not been adequately addressed by previous techniques.

We demonstrate the effectiveness of our approach on six popular open source PHP code bases, finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

1 Introduction

Web-based applications have proliferated rapidly in recent years and have become the *de facto* standard for delivering online services ranging from discussion forums to security sensitive areas such as banking and retailing. As such, security vulnerabilities in these applications represent an increasing threat to both the providers and the users of such services. During the second half of 2004, Symantec cataloged 670 vulnerabilities affecting web applications, an 81% increase over the same period in 2003 [17]. This trend is likely to continue for the foreseeable future.

According to the same report, these vulnerabilities are typically caused by programming errors in input validation and improper handling of submitted requests [17]. Since vulnerabilities are usually deeply embedded in the program logic, traditional network-level defense (e.g., firewalls) does not offer adequate protection against such attacks. Testing is also largely ineffective because attackers typically use the least expected input to exploit these vulnerabilities and compromise the system.

A natural alternative is to find these errors using static analysis. This approach has been explored in WebSSARI [7] and by Minamide [10]. WebSSARI has been used to find a number of security vulnerabilities in PHP scripts, but has a large number of false positives and negatives due to its intraprocedural type-based analysis. Minamide's system checks syntactic correctness of HTML output from PHP scripts and does not seem to be effective for finding security vulnerabilities. The main message of this paper is that analysis of scripting languages need not be significantly more difficult than analysis of conventional languages. While a scripting language stresses different aspects of static analysis, an analysis suitably designed to address the important aspects of scripting languages can identify many serious vulnerabilities in scripts reliably and with a high degree of automation. Given the importance of scripting in real world applications, we believe there is an opportunity for static analysis to have a significant impact in this new domain.

In this paper, we apply static analysis to finding security vulnerabilities in PHP, a server-side scripting language that has become one of the most widely adopted platforms for developing web applications.[1] Our goal is a bug detection tool that automatically finds serious vulnerabilities with high confidence. This work, however, does not aim to verify the absence of bugs.

This paper makes the following contributions:

- We present an interprocedural static analysis algorithm for PHP. A language as dynamic as PHP presents unique challenges for static analysis: language constructs (e.g., include) that allow dynamic inclusion of program code, variables whose types change during execution, operations with semantics that depend on the runtime types of the operands (e.g., <), and pervasive use of hash tables and regular expression matching are just some features that must be modeled well to produce useful results.

[1] Installed on over 23 million Internet domains [14], and is ranked fourth on the TIOBE programming community index [18].

To faithfully model program behavior in such a language, we use a three-tier analysis that captures information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural levels. This architecture allows the analysis to be precise where it matters the most–at the intrablock and, to a lesser extent, the intraprocedural levels–and use agressive abstraction at the natural abstraction boundary along function calls to achieve scalability. We use symbolic execution to model dynamic features inside basic blocks and use block summaries to hide that complexity from intra- and inter-procedural analysis. We believe the same techniques can be applied easily to other scripting languages (e.g., Perl).

- We show how to use our static analysis algorithm to find SQL injection vulnerabilities. Once configured, the analysis is fully automatic. Although we focus on SQL injections in this work, the same techniques can be applied to detecting other vulnerabilities such as cross site scripting (XSS) and code injection in web applications.
- We experimentally validate our approach by implementing the analysis algorithm and running it on six popular web applications written in PHP, finding 105 previously unknown security vulnerabilities. We analyzed two reported vulnerabilities in PHP-fusion, a mature, widely deployed content management system, and construct exploits for both that allow an attacker to control or damage the system.[2]

The rest of the paper is organized as follows. We start with a brief introduction to PHP and show examples of SQL vulnerabilities in web application code (Section 2). We then present our analysis algorithm and show how we use it to find SQL injection vulnerabilities (Section 3). Section 4 describes the implementation, experimental results, and two case studies of exploitable vulnerabilities in PHP-fusion. Section 5 discusses related work and Section 6 concludes.

2 Background

This section briefly introduces the PHP language and shows examples of SQL injection vulnerabilities in PHP.

PHP was created a decade ago by Rasmus Lerdorf as a simple set of Perl scripts for tracking accesses to his online resume. It has since evolved into one of the most popular server-side scripting languages for building web applications. According to a recent Security Space survey, PHP is installed on 44.6% of Apache web servers [16], adopted by millions of developers, and used or supported by Yahoo, IBM, Oracle, and SAP, among others [14].

Although the PHP language has undergone two major re-designs over the past decade, it retains a Perl-like syntax and dynamic (interpreted) nature, which contributes to its most frequently claimed advantage of being simple and flexible.

PHP has a suite of programming constructs and special operations that ease web development. We give three examples:

1. **Natural integration with SQL:** PHP provides nearly native support for database operations. For example, using inline variables in strings, most SQL queries can be concisely expressed with a simple function call

   ```
   $rows=mysql_query("UPDATE users SET
       pass='$pass' WHERE userid='$userid'");
   ```

   Contrast this code with Java, where a database is typically accessed through *prepared statements*: one creates a statement template and fills in the values (along with their types) using *bind variables*:

   ```
   PreparedStatement s = con.prepareStatement
     ("UPDATE users SET pass = ?
       WHERE userid = ?");
   s.setString(1, pass); s.setInt(2, userid);
   int rows = s.executeUpdate();
   ```

2. **Dynamic types and implicit casting to and from strings:** PHP, like other scripting languages, has extensive support for string operations and automatic conversions between strings and other types. These features are handy for web applications because strings serve as the common medium between the browser, the web server, and the database backend. For example, we can convert a number into a string without an explicit cast:

   ```
   if ($userid < 0) exit;
   $query = "SELECT * from users
             WHERE userid = '$userid'";
   ```

3. **Variable scoping and the environment:** PHP has a number of mechanisms that minimize redundancy when accessing values from the execution environment. For example, HTTP *get* and *post* requests are automatically imported into the global name space as hash tables `$_GET` and `$_POST`. To access the "name" field of a submitted form, one can simply use `$_GET['name']` directly in the program.

   If this still sounds like too much typing, PHP provides an `extract` operation that automatically imports all key-value pairs of a hash table into the current scope. In the example above, one can

[2] Both vulnerabilities have been reported to and fixed by the PHP-fusion developers.

use extract(_GET, EXTR_OVERWRITE) to import data submitted using the HTTP `get` method. To access the $name field, one now simply types $name, which is preferred by some to $_GET['name'].

However, these conveniences come with security implications:

1. **SQL injection made easy:** Bind variables in Java have the benefit of assuring the programmer that any data passed into an SQL query remains data. The same cannot be said for the PHP example where malformed data from a malicious attacker may change the meaning of an SQL statement and cause unintended operations to the database. These are commonly called *SQL injection* attacks.

   In the example above (case 1), suppose $userid is controlled by the attacker and has value

   ```
   ' OR '1' = '1
   ```

   The query string becomes

   ```
   UPDATE users SET pass='...'
   WHERE userid='' OR '1'='1'
   ```

   which has the effect of updating the password for all users in the database.

2. **Unexpected conversions:** Consider the following code:

   ```
   if ($userid == 0) echo $userid;
   ```

   One would expect that if the program prints anything, it should be "0". Unfortunately, PHP implicitly casts string values into numbers before comparing them with an integer. Non-numerical values (e.g., "abc") convert to 0 without complaint, so the code above can print anything other than a non-zero number. We can imagine a potential SQL injection vulnerability if $userid is subsequently used to construct an SQL query as in the previous case.

3. **Uninitialized variables under user control:** In PHP, uninitialized variables default to null. Some programs rely on this fact for correct behavior; consider the following code:

   ```
   1  extract($_GET, EXTR_OVERWRITE);
   2  for ($i=0;$i<=7;$i++)
   3    $new_pass .= chr(rand(97, 122)); // append one char
   4  mysql_query("UPDATE ... $new_pass ...");
   ```

   This program generates a random password and inserts it into the database. However, due to the extract operation on line 1, a malicious user can introduce an arbitrary initial value for $new_pass by adding an unexpected new_pass field into the submitted HTTP form data.

```
CFG := build_control_flow_graph(AST);
foreach (basic_block b in CFG)
  summaries[b] := simulate_block(b);
return make_function_summary(CFG, summaries);
```

Figure 1: Pseudo-code for the analysis of a function.

3 Analysis

Given a PHP source file, our tool carries out static analysis in the following steps:

- We parse the PHP source into abstract syntax trees (ASTs). Our parser is based on the standard open-source implementation of PHP 5.0.5 [13]. Each PHP source file contains a *main* section (referred to as the *main* function hereafter although it is not part of any function definition) and zero or more user-defined functions. We store the user-defined functions in the environment and start the analysis from the main function.

- The analysis of a single function is summarized in Figure 1. For each function in the program, the analysis performs a standard conversion from the abstract syntax tree (AST) of the function body into a control flow graph (CFG). The nodes of the CFG are *basic blocks*: maximal single entry, single exit sequences of statements. The edges of the CFG are the jump relationships between blocks. For conditional jumps, the corresponding CFG edge is labeled with the branch predicate.

- Each basic block is simulated using symbolic execution. The goal is to understand the collective effects of statements in a block on the global state of the program and summarize their effects into a concise *block summary* (which describes, among other things, the set of variables that must be sanitized[3] before entering the block). We describe the simulation algorithm in Section 3.1.

- After computing a summary for each basic block, we use a standard reachability analysis to combine block summaries into a *function summary*. The function summary describes the pre- and post-conditions of a function (e.g., the set of sanitized input variables after calling the current function). We discuss this step in Section 3.2.

- During the analysis of a function, we might encounter calls to other user-defined functions. We discuss modeling function calls, and the order in which functions are analyzed, in Section 3.3.

[3] Sanitization is an operation that ensures that user input can be safely used in an SQL query (e.g., no unescaped quotes or spaces).

```
function simulate_block(BasicBlock b) : BlockSummary
{
  state := init_simulation_state();
  foreach (Statement s in b) {
    state := simulate(s, state);
    if (state.has_returned || state.has_exited)
      break;
  }
  summary := make_block_summary(state);
  return summary;
}
```

Figure 2: Pseudo-code for intra-block simulation.

$$
\begin{array}{rl}
\text{Type } (\tau) & ::= \text{str} \mid \text{bool} \mid \text{int} \mid \top \\
\text{Const } (c) & ::= \text{string} \mid \text{int} \mid \text{true} \mid \text{false} \mid \text{null} \\
\text{L-val } (lv) & ::= x \mid \texttt{Arg\#i} \mid lv[e] \\
\text{Expr } (e) & ::= c \mid lv \mid e \text{ binop } e \mid \text{unop } e \mid (\tau)e \\
\text{Stmt } (S) & ::= lv \leftarrow e \mid lv \leftarrow f(e_1, \ldots, e_n) \\
 & \quad \mid \text{return } e \mid \text{exit} \mid \text{include } e
\end{array}
$$

$$
\begin{array}{rl}
\text{binop} & \in \{+, -, \text{concat}, ==, !=, <, >, \ldots\} \\
\text{unop} & \in \{-, \neg\}
\end{array}
$$

Figure 3: Language Definition

3.1 Simulating Basic Blocks

3.1.1 Outline

Figure 2 gives pseudo-code outlining the symbolic simulation process. Recall each basic block contains a linear sequence of statements with no jumps or jump targets in the middle. The simulation starts in an *initial state*, which maps each variable x to a symbolic initial value x_0. It processes each statement in the block in order, updating the simulator state to reflect the effect of that statement. The simulation continues until it encounters any of the following:

1. the end of the block;
2. a return statement. In this case, the current block is marked as a *return* block, and the simulator evaluates and records the return value;
3. an exit statement. In this case the current block is marked as an *exit* block;
4. a call to a user-defined function that exits the program. This condition is automatically determined using the function summary of the callee (see Sections 3.2 and 3.3).

Note that in the last case execution of the program has also terminated and therefore we remove any ensuing statements and outgoing CFG edges from the current block.

After a basic block is simulated, we use information contained in the final state of the simulator to summarize the effect of the block into a *block summary*, which we store for use during the intraprocedural analysis (see Section 3.2). The state itself is discarded after simulation.

The following subsections describe the simulation process in detail. We start with a definition of the subset of PHP that we model (§3.1.2) and discuss the representation of the simulation state and program values (§3.1.3, §3.1.4) during symbolic execution. Using the value representation, we describe how the analyzer simulates expressions (§3.1.5) and statements (§3.1.6). Finally, we describe how we represent and infer block summaries (§3.1.7).

3.1.2 Language

Figure 3 gives the definition of a small imperative language that captures a subset of PHP constructs that we believe is relevant to SQL injection vulnerabilities. Like PHP, the language is dynamically typed. We model three basic types of PHP values: strings, booleans and integers. In addition, we introduce a special $\top$ type to describe objects whose static types are undetermined (e.g., input parameters).[4]

Expressions can be *constants*, *l-values*, *unary* and *binary operations*, and *type casts*. The definition of l-values is worth mentioning because in addition to variables and function parameters, we include a named subscript operation to give limited support to the array and hash table accesses used extensively in PHP programs.

A statement can be an *assignment*, *function call*, *return*, *exit*, or *include*. The first four statement types require no further explanation. The include statement is a commonly used feature unique to scripting languages, which allows programmers to dynamically insert code into the program. In our language, include evaluates its string argument, and executes the program file designated by the string as if it is inserted at that program point (e.g., it shares the same scope). We describe how we simulate such behavior in Section 3.1.6.

3.1.3 State

Figure 4(a) gives the definition of values and states during simulation. The simulation state maps memory locations to their value representations, where a memory location is either a program variable (e.g. x), or an entry in a hash table accessed via another location (e.g. $x[key]$). Note the definition of locations is recursive, so multi-level hash dereferences are supported in our algorithm.

[4] In general, in a dynamically typed language, a more precise static approximation in this case would be a sum (aka. soft typing) [1, 20]. We have not found it necessary to use type sums in this work.

$\boxed{\text{Value Representation}}$

$$\begin{aligned}
\text{Loc } (l) &::= x \mid l[\text{string}] \mid l[\top] \\
\text{Init-Values } (o) &::= l_0 \\
\text{Segment } (\beta) &::= \text{string} \mid \text{contains}(\sigma) \mid o \mid \bot \\
\text{String } (s) &::= \langle \beta_1, \ldots, \beta_n \rangle \\
\text{Boolean } (b) &::= \text{true} \mid \text{false} \mid \text{untaint}(\sigma_0, \sigma_1) \\
\text{Loc-set}(\sigma) &::= \{l_1, \ldots, l_n\} \\
\text{Integer } (i) &::= k \\
\text{Value } (v) &::= s \mid b \mid i \mid o \mid \top
\end{aligned}$$

$\boxed{\text{Simulation State}}$

$$\text{State } (\Gamma) : \text{Loc} \rightarrow \text{Value}$$

(a) Value representation and simulation state.

$\boxed{\text{Locations}}$

$$\frac{}{\Gamma \vdash x \overset{\text{Lv}}{\Rightarrow} x}\ \text{var} \qquad \frac{}{\Gamma \vdash \texttt{Arg\#n} \overset{\text{Lv}}{\Rightarrow} \texttt{Arg\#n}}\ \text{arg}$$

$$\frac{\begin{array}{c} \Gamma \vdash e \overset{\text{E}}{\Rightarrow} l \\ \Gamma \vdash e' \overset{\text{E}}{\Rightarrow} v' \quad v'' = \text{cast}(v', \text{str}) \end{array}}{\Gamma \vdash e[e'] \overset{\text{Lv}}{\Rightarrow} \begin{cases} l[\alpha] & \text{if } v'' = \langle \text{“}\alpha\text{”} \rangle \\ l[\top] & \text{otherwise} \end{cases}}\ \text{dim}$$

(b) L-values.

$\boxed{\text{Expressions}}$

Type casts:

$$\text{cast}(k, \text{bool}) = \begin{cases} \text{true} & \text{if } k \neq 0 \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{cast}(\text{true}, \text{str}) = \langle \text{“1”} \rangle$$

$$\text{cast}(\text{false}, \text{str}) = \langle \rangle$$

$$\begin{aligned} &\text{cast}(v = \langle \beta_1, \ldots, \beta_n \rangle, \text{bool}) \\ &\quad = \begin{cases} \text{true} & \text{if } (v \neq \langle \text{“0”} \rangle) \wedge \bigvee_{i=1}^{n} \neg\text{is\_empty}(\beta_i) \\ \text{false} & \text{if } (v = \langle \text{“0”} \rangle) \vee \bigwedge_{i=1}^{n} \text{is\_empty}(\beta_i) \\ \top & \text{otherwise} \end{cases} \end{aligned}$$

...

Evaluation Rules:

$$\frac{\Gamma \vdash lv \overset{\text{Lv}}{\Rightarrow} l}{\Gamma \vdash lv \overset{\text{E}}{\Rightarrow} \Gamma(l)}\ \text{L-val}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 \overset{\text{E}}{\Rightarrow} v_1 \quad \text{cast}(v_1, \text{str}) = \langle \beta_1, \ldots, \beta_n \rangle \\ \Gamma \vdash e_2 \overset{\text{E}}{\Rightarrow} v_2 \quad \text{cast}(v_2, \text{str}) = \langle \beta_{n+1}, \ldots, \beta_m \rangle \end{array}}{\Gamma \vdash e_1 \text{ concat } e_2 \overset{\text{E}}{\Rightarrow} \langle \beta_1, \ldots, \beta_m \rangle}\ \text{concat}$$

$$\frac{\Gamma \vdash e \overset{\text{E}}{\Rightarrow} v \quad \text{cast}(v, \text{bool}) = v'}{\Gamma \vdash \neg e \overset{\text{E}}{\Rightarrow} \begin{cases} \text{true} & \text{if } v' = \text{false} \\ \text{false} & \text{if } v' = \text{true} \\ \text{untaint}(\sigma_1, \sigma_0) & \text{if } v' = \text{untaint}(\sigma_0, \sigma_1) \\ \top & \text{otherwise} \end{cases}}\ \text{not}$$

(c) Expressions.

Figure 4: Intrablock simulation algorithm.

On entry to the function, each location l is implicitly initialized to a symbolic initial value l_0, which makes up the initial state of the simulation. The values we represent in the state can be classified into three categories based on type:

Strings: Strings are the most fundamental type in many scripting languages, and precision in modeling strings directly determines analysis precision. Strings are typically constructed through concatenation. For example, user inputs (via HTTP `get` and `post` methods) are often concatenated with a pre-constructed skeleton to form an SQL query. Similarly, results from the query can be concatenated with HTML templates to form output. Modeling concatenation well enables an analysis to better understand information flow in a script. Thus, our string representations is based on concatenation. String values are represented as an ordered concatenation of string *segments*, which can be one of the following: a string constant, the initial value of a memory location on entry to the current block (l_0), or a string that contains initial values of zero or more elements from a set of memory locations (contains(σ)). We use the last representation to model return values from function calls, which may non-deterministically contain a combination of global variables and input parameters. For example, in

```
1  function f($a, $b) {
2      if (...) return $a;
3      else return $b;
4  }
5  $ret = f($x.$y, $z);
```

we represent the return value on line 5 as contains$(\{\text{x}, \text{y}, \text{z}\})$ to model the fact that it may contain any element in the set as a sub-string.

The string representation described above has the following benefits:

First, we get automatic constant folding for strings within the current block, which is often useful for resolving hash keys and distinguishing between hash references (e.g., in $key = “key”; return $hash[$key];).

Second, we can track how the contents of one input variable flow into another by finding occurrences of initial values of the former in the final representation of the latter. For example, in: $a = $a . $b, the final representation of $a is $\langle a_0, b_0 \rangle$. We know that if either $a or $b contains unsanitized user input on entry to the current block, so does $a upon exit.

Finally, interprocedural dataflow is possible by tracking function return values based on function summaries using contains(σ). We describe this aspect in more detail in Section 3.3.

Booleans: In PHP, a common way to perform input validation is to call a function that returns true or false depending on whether the input is well-formed or not. For example, the following code sanitizes $userid:

```
$ok = is_safe($userid);
if (!$ok) exit;
```

The value of Boolean variable $ok after the call is undetermined, but it is correlated with the safety of $userid. This motivates $\mathsf{untaint}(\sigma_0, \sigma_1)$ as a representation for such Booleans: σ_0 (resp. σ_1) represents the set of validated l-values when the Boolean is false (resp. true). In the example above, $ok has representation $\mathsf{untaint}(\{\}, \{\mathsf{userid}\})$.

Besides untaint, representation for Booleans also include constants (true and false) and unknown ($\top$).

Integers: Integer operations are less emphasized in our simulation. We track integer constants and binary and unary operations between them. We also support type casts from integers to Boolean and string values.

3.1.4 Locations and L-values

In the language definition in Figure 3, hash references may be aliased through assignments and l-values may contain hash accesses with non-constant keys. The same l-value may refer to different memory locations depending on the value of both the host and the key, and therefore, l-values are not suitable as memory locations in the simulation state.

Figure 4(b) gives the rules we use to resolve l-values into memory locations. The var and arg rules map each program variable and function argument to a memory location identified by its name, and the dim rule resolves hash accesses by first evaluating the hash table to a location and then appending the key to form the location for the hash entry.

These rules are designed to work in the presence of simple aliases. Consider the following program:

```
1  $hash = $_POST;
2  $key = 'userid';
3  $userid = $hash[$key];
```

The program first creates an alias ($hash) to hash table $_POST and then accesses the userid entry using that alias. On entry to the block, the initial state maps every location to its initial value:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{hash}_0, \mathsf{key} \Rightarrow \mathsf{key}_0, \mathsf{\_POST} \Rightarrow \mathsf{\_POST}_0, \\ \mathsf{\_POST[userid]} \Rightarrow \mathsf{\_POST[userid]}_0\}$$

According to the var rule, each variable maps to its own unique location. After the first two assignments, the state is:

$$\Gamma = \{\mathsf{hash} \Rightarrow \mathsf{\_POST}_0, \mathsf{key} \Rightarrow \langle\text{‘userid’}\rangle, \ldots\}$$

We use the dim rule to resolve $hash[$key] on line 3: $hash evaluates to $\mathsf{\_POST}_0$, and $key evaluates to constant string 'userid'. Therefore, the l-value $hash[$key] evaluates to location _POST[userid], and thus the analysis assigns the desired value $\mathsf{\_POST[userid]}_0$ to $userid.

3.1.5 Expressions

We perform abstract evaluation of expressions based on the value representation described above. Because PHP is a dynamically typed language, operands are implicitly cast to appropriate types for operations in an expression. Figure 4(c) gives a representative sample of cast rules simulating cast operations in PHP. For example, Boolean value true, when used in a string context, evaluates to “1”. false, on the other hand, is converted to the empty string instead of “0”. In cases where exact representation is not possible, the result of the cast is unknown ($\top$).

Figure 4(c) also gives three representative rules for evaluating expressions. The first rule handles l-values, and the result is obtained by first resolving the l-value into a memory location, and then looking up the location in the evaluation context (recall that $\Gamma(l) = l_0$ on entry to the block).

The second rule models string concatenation. We first cast the value of both operands to string values, and the result is the concatenation of both.

The final rule handles Boolean negation. The interesting case involves untaint values. Recall that $\mathsf{untaint}(\sigma_0, \sigma_1)$ denotes an unknown Boolean value that is false (resp. true) if l-values in the set σ_0 (resp. σ_1) are sanitized. Given this definition, the negation of $\mathsf{untaint}(\sigma_0, \sigma_1)$ is $\mathsf{untaint}(\sigma_1, \sigma_0)$.

The analysis of an expression is $\top$ if we cannot determine a more precise representation, which is a potential source of false negatives.

3.1.6 Statements

We model assignments, function calls, return, exit, and include statements in the program. The assignment rule resolves the left-hand side to a memory location l, and evaluates the right-hand side to a value v. The updated simulation state after the assignment maps l to the new value v:

$$\frac{\Gamma \vdash lv \overset{\mathtt{Lv}}{\Rightarrow} l \qquad \Gamma \vdash e \overset{\mathtt{E}}{\Rightarrow} v}{\Gamma \vdash lv \leftarrow e \overset{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto v]} \ \mathsf{assignment}$$

Function calls are similar. The return value of a function call $f(e_1, \ldots, e_n)$ is modeled using either $\mathsf{contains}(\sigma)$ (if f returns a string) or $\mathsf{untaint}(\sigma_0, \sigma_1)$ (if f returns a Boolean) depending on the inferred summary for f. We defer discussion of the function summaries and the return value representation to Sections 3.2 and 3.3. For the purpose of this section, we use the uninterpreted value $f(v_1, \ldots, v_n)$ as a place holder for the actual representation of the return value:

$$\frac{\Gamma \vdash lv \overset{\mathtt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\mathtt{E}}{\Rightarrow} v_1 \ \ldots \ \Gamma \vdash e_n \overset{\mathtt{E}}{\Rightarrow} v_n}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\mathtt{S}}{\Rightarrow} \Gamma[l \mapsto f(v_1, \ldots, v_n)]} \ \mathsf{fun}$$

In addition to the return value, certain functions have pre- and post-conditions depending on the operation they perform. Pre- and post-conditions are inferred and stored in the callee's summary, which we describe in detail in Sections 3.2 and 3.3. Here we show two examples to illustrate their effects:

```
1  function validate($x) {
2    if (!is_numeric($x)) exit;
3    return;
4  }
5  function my_query($q) {
6     global $db;
7     mysql_db_query($db, $q);
8  }
9  validate($a.$b);
10 my_query("SELECT. . .WHERE a = '$a' AND c = '$c'");
```

The validate function tests whether the argument is a number (and thus safe) and aborts if it is not. Therefore, line 9 sanitizes both $a and $b. We record this fact by inspecting the value representation of the actual parameter (in this case $\langle a_0, b_0 \rangle$), and remembering the set of non-constant segments that are sanitized.

The second function my_query uses its argument as a database query string by calling mysql_db_query. To prevent SQL injection attacks, any user input must be sanitized before it becomes part of the first parameter. Again, we enforce this requirement by inspecting the value representation of the actual parameter. We record any unsanitized non-constant segments (in this case $c, since $a is sanitized on line 9) and require they be sanitized as part of the pre-condition for the current block.

Sequences of assignments and function calls are simulated by using the output environment of the previous statement as the input environment of the current statement:

$$\frac{\Gamma \vdash s_1 \overset{S}{\Rightarrow} \Gamma' \quad \Gamma' \vdash s_2 \overset{S}{\Rightarrow} \Gamma''}{\Gamma \vdash (s_1; s_2) \overset{S}{\Rightarrow} \Gamma''}\ \text{seq}$$

The final simulation state is the output state of the final statement.

The return and exit statements terminate control flow[5] and require special treatment. For a return, we evaluate the return value and use it in calculating the function summary. In case of an exit statement, we mark the current block as an *exit block*.

Finally, include statements are a commonly used feature unique to scripting languages allowing programmers to dynamically insert code and function definitions from another script. In PHP, the included code inherits the variable scope at the point of the include statement. It may introduce new variables and function definitions, and change or sanitize existing variables before the next statement in the block is executed.

We process include statements by first parsing the included file, and adding any new function definitions to the environment. We then splice the control flow graph of the included main function at the current program point by a) removing the include statement, b) breaking the current basic block into two at that point, c) linking the first half of the current block to the start of the main function, and all return blocks (those containing a return statement) in the included CFG to the second half, and d) replacing the return statements in the included script with assignments to reflect the fact that control flow resumes in the current script.

3.1.7 Block summary

The final step for the symbolic simulator is to characterize the behavior of a CFG block into a concise summary. A block summary is represented as a six-tuple $\langle \mathcal{E}, \mathcal{D}, \mathcal{F}, \mathcal{T}, \mathcal{R}, \mathcal{U} \rangle$:

- **Error set ($\mathcal{E}$):** the set of input variables that must be sanitized before entering the current block. These are accumulated during simulation of function calls that require sanitized input.
- **Definitions ($\mathcal{D}$):** the set of memory locations defined in the current block. For example, in

  ```
  $a = $a.$b; $c = 123;
  ```

  we have $\mathcal{D} = \{a, c\}$.
- **Value flow ($\mathcal{F}$):** the set of pairs of locations (l_1, l_2) where the string value of l_1 on entry becomes a substring of l_2 on exit. In the example above, $\mathcal{F} = \{(a, a), (b, a)\}$.
- **Termination predicate ($\mathcal{T}$):** true if the current block contains an exit statement, or if it calls a function that causes the program to terminate.
- **Return value ($\mathcal{R}$):** records the representation for the return value if any, undefined otherwise. Note that if the current block has no successors, either $\mathcal{R}$ has a value or $\mathcal{T}$ is true.
- **Untaint set ($\mathcal{U}$):** for each successor of the current CFG block, we compute the set of locations that are sanitized if execution continues onto that block. Sanitization can occur via function calls, casting to safe types (e.g., int, etc), regular expression matching, and other tests. The untaint set for different successors might differ depending on the value of branch predicates. We show an example below.

  ```
  validate($a);
  $b = (int) $c;
  if (is_numeric($d))
      . . .
  ```

  As mentioned earlier, validate exits if $a is unsafe. Casting to integer also returns a safe result. Therefore, the untaint set is $\{a, b, d\}$ for the true branch, and $\{a, b\}$ for the false branch.

[5] So do function calls that exits the program, in which case we remove any ensuing statements and outgoing edges from the current CFG block. See Section 3.3.

3.2 Intraprocedural Analysis

Based on block summaries computed in the previous step, the intraprocedural analysis computes the following summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$ for each function:

1. **Error set ($\mathcal{E}$):** the set of memory locations (variables, parameters, and hash accesses) whose value may flow into a database query, and therefore must be sanitized before invoking the current function. For the main function, the error set must not include any user-defined variables (e.g. $_GET['...'] or $_POST['...'])—the analysis emits an error message for each such violation.

   We compute $\mathcal{E}$ by a backwards reachability analysis that propagates the error set of each block (using the $\mathcal{E}, \mathcal{D}, \mathcal{F}$, and $\mathcal{U}$ components in the block summaries) to the start block of the function.

2. **Return set ($\mathcal{R}$):** the set of parameters or global variables whose value may be a substring of the return value of the function. $\mathcal{R}$ is only computed for functions that may return string values. For example, in the following code, the return set includes both function arguments and the global variable $table (i.e., $\mathcal{R} = \{\mathsf{table}, \mathtt{Arg\#1}, \mathtt{Arg\#2}\}$).

   ```
   function make_query($user, $pass) {
     global $table;
     return "SELECT * from $table ".
       "where user = $user and pass = $pass";
   }
   ```

   We compute the function return set by using a forward reachability analysis that expresses each return value (recorded in the block summaries as $\mathcal{R}$) as a set of function parameters and global variables.

3. **Sanitized values ($\mathcal{S}$):** the set of parameters or global variables that are sanitized on function exit. We compute the set by using a forward reachability analysis to determine the set of sanitized inputs at each return block, and we take the intersection of those sets to arrive at the final result.

   If the current function returns a Boolean value as its result, we distinguish the sanitized value set when the result is true versus when it is false (mirroring the untaint representation for Boolean values above). The following example motivates this distinction:

   ```
   function is_valid($x) {
     if (is_numeric($x)) return true;
     return false;
   }
   ```

   The parameter is sanitized if the function returns true, and the return value is likely to be used by the caller to determine the validity of user input. In the example above,

   $$\mathcal{S} = (\mathsf{false} \Rightarrow \{\}, \mathsf{true} \Rightarrow \{\mathtt{Arg\#1}\})$$

   For comparison, the validate function defined previously has $\mathcal{S} = (* \Rightarrow \{\mathtt{Arg\#1}\})$. In the next section, we describe how we make use of this information in the caller.

4. **Program Exit ($\mathcal{X}$):** a Boolean which indicates whether the current function terminates program execution on all paths. Note that control flow can leave a function either by returning to the caller or by terminating the program. We compute the exit predicate by enumerating over all CFG blocks that have no successors, and identify them as either return blocks or exit blocks (the $\mathcal{T}$ and $\mathcal{R}$ component in the block summary). If there are no return blocks in the CFG, the current function is an exit function.

The dataflow algorithms used in deriving these facts are fairly standard fix-point computations. We omit the details for brevity.

3.3 Interprocedural Analysis

This section describes how we conduct interprocedural analysis using summaries computed in the previous step. Assuming f has summary $\langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle$, we process a function call $f(e_1, \ldots, e_n)$ during intrablock simulation as follows:

1. **Pre-conditions:** We use the error set ($\mathcal{E}$) in the function summary to identify the set of parameters and global variables that must be sanitized before calling this function. We substitute actual parameters for formal parameters in $\mathcal{E}$ and record any unsanitized non-constant segments of strings in the error set as the sanitization pre-condition for the current block.

2. **Exit condition:** If the callee is marked as an exit function (i.e., $\mathcal{X}$ is true), we remove any statements that follow the call and delete all outgoing edges from the current block. We further mark the current block as an exit block.

3. **Post-conditions:** If the function unconditionally sanitizes a set of input parameters and global variables, we mark this set of values as safe in the simulation state after substituting actual parameters for formal parameters.

   If sanitization is conditional on the return value (e.g., the is_valid function defined above), we record the intersection of its two component sets as being unconditionally sanitized (i.e., $\sigma_0 \cap \sigma_1$ if the untaint set is $(\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1)$).

4. **Return value:** If the function returns a Boolean value and it conditionally sanitizes a set of input parameters and global variables, we use the `untaint` representation to model that correlation:

$$\frac{\begin{array}{l}\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots\ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n \\ \texttt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \\ \mathcal{S} = (\mathsf{false} \Rightarrow \sigma_0, \mathsf{true} \Rightarrow \sigma_1) \quad \sigma_* = \sigma_0 \cap \sigma_1 \\ \sigma_0' = \mathsf{subst}_{\bar{v}}(\sigma_0 - \sigma_*) \quad \sigma_1' = \mathsf{subst}_{\bar{v}}(\sigma_1 - \sigma_*)\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{untaint}(\sigma_0', \sigma_1')]}$$

In the rule above, $\mathsf{subst}_{\bar{v}}(\sigma)$ substitutes actual parameters (v_i) for formal parameters in σ.

If the callee returns a string value, we use the return set component of the function summary ($\mathcal{R}$) to determine the set of input parameters and global variables that might become a substring of the return value:

$$\frac{\begin{array}{l}\Gamma \vdash lv \overset{\texttt{Lv}}{\Rightarrow} l \quad \Gamma \vdash e_1 \overset{\texttt{E}}{\Rightarrow} v_1 \ \ldots\ \Gamma \vdash e_n \overset{\texttt{E}}{\Rightarrow} v_n \\ \texttt{Summary}(f) = \langle \mathcal{E}, \mathcal{R}, \mathcal{S}, \mathcal{X} \rangle \quad \sigma' = \mathsf{subst}_{\bar{v}}(\mathcal{R})\end{array}}{\Gamma \vdash lv \leftarrow f(e_1, \ldots, e_n) \overset{\texttt{S}}{\Rightarrow} \Gamma[l \mapsto \mathsf{contains}(\sigma')]}$$

Since we require the summary information of a function before we can analyze its callers, the order in which functions are analyzed is important. Due to the dynamic nature of PHP (e.g., `include` statements), we analyze functions on demand—a function f is analyzed and summarized when we first encounter a call to f. The summary is then memoized to avoid redundant analysis. Effectively, our algorithm analyzes the source codebase in topological order based on the static function call graph. If we encounter a cycle during the analysis, the current implementation uses a dummy "no-op" summary as a model for the second invocation (i.e., we do not compute fix points for recursive functions). In theory, this is a potential source of false negatives, which can be removed by adding a simple iterative algorithm that handles recursion. However, practically, such an algorithm may be unnecessary given the rare occurrence of recursive calls in PHP programs.

4 Experimental Results

The analysis described in Section 3 has been implemented as two separate parts: a frontend based on the open source PHP 5.0.5 distribution that parses the source files into abstract syntax trees and a backend written in OCaml [8] that reads the ASTs into memory and carries out the analysis. This separation ensures maximum compatibility while minimizing dependence on the PHP implementation.

The decision to use different levels of abstraction in the intrablock, intraprocedural, and interprocedural levels enabled us to fine tune the amount of information we retain at one level independent of the algorithm used in another and allowed us to quickly build a usable tool. The checker is largely automatic and requires little human intervention for use. We seed the checker with a small set of query functions (e.g. mysql_query) and sanitization operations (e.g. is_numeric). The checker infers the rest automatically.

Regular expression matching presents a challenge to automation. Regular expressions are used for a variety of purposes including, but not limited to, input validation. Some regular expressions match well-formed input while others detect malformed input; assuming one way or the other results in either false positives or false negatives. Our solution is to maintain a database of previously seen regular expressions and their effects, if any. Previously unseen regular expressions are assumed by default to have no sanitization effects, so as not to miss any errors due to incorrect judgment. To make it easy for the user to specify the sanitization effects of regular expressions, the checker has an interactive mode where the user is prompted when the analysis encounters a previously unseen regular expression and the user's answers are recorded for future reference.[6] Having the user declare the role of regular expressions has the real potential to introduce errors into the analysis; however, practically, we found this approach to be very effective and it helped us find at least two vulnerabilities caused by overly lenient regular expressions being used for sanitization.[7] Our tool collected information for 49 regular expressions from the user over all our experiments (the user replies with one keystroke for each inquiry), so the burden on the user is minimal.

The checker detects errors by using information from the summary of the main function—the checker marks all variables that are required to be sanitized on entry as potential security vulnerabilities. From the checker's perspective, these variables are defined in the environment and used to construct SQL queries without being sanitized. In reality, however, these variables are either defined by the runtime environment or by some language constructs that the checker does not fully understand (e.g., the `extract` operation in PHP which we describe in a case study below). The tool emits an *error* mes-

[6] Here we assume that a regular expression used to sanitize input in one context will have the same effect in another, which, based on our experience, is the common case. Our implementation now provides paranoid users with a special switch that ignores recorded answers and repeatedly ask the user the same question over and over if so desired.

[7] For example, Utopia News Pro misused "`[0-9]+`" to validate some user input. This regular expression only checks that the string contains a number, instead of ensuring that the input *is* actually a number. The correct regular expression in this case is "`^[0-9]+$`".

Application (KLOC)	**Err Msgs**	**Bugs (FP)**	**Warn**
News Pro (6.5)	8	8 (0)	8
myBloggie (9.2)	16	16 (0)	23
PHP Webthings (38.3)	20	20 (0)	6
DCP Portal (121)	39	39 (0)	55
e107 (126)	16	16 (0)	23
Total	99	99 (0)	115

Table 1: Summary of experiments. LOC statistics include embedded HTML, and thus is a rough estimate of code complexity. Err Msgs: number of reported errors. Bugs: number of confirmed bugs from error reports. FP: number of false positives. Warn: number of unique warning messages for variables of unresolved origin (uninspected).

sage if the variable is known to be controlled by the user (e.g. $_GET['...'], $_POST['...'], $_COOKIE['...'], etc). For others, the checker emits a *warning*.

We conducted our experiments on the latest versions of six open source PHP code bases: `e107 0.7`, `Utopia News Pro 1.1.4`, `mybloggie 2.1.3beta`, `DCP Portal v6.1.1`, `PHP Webthings 1.4patched`, and `PHP fusion 6.00.204`. Table 1 summarizes our findings for the first five. The analysis terminates within seconds for each script examined (which may dynamically include other source files). Our checker emitted a total of 99 error messages for the first five applications, where unsanitized user input (from $_GET, $_POST, etc) may flow into SQL queries. We manually inspected the error reports and believe all 99 represent real vulnerabilities.[8] We have notified the developers about these errors and will publish security advisories once the errors have been fixed. We have not inspected warning messages—unsanitized variables of unresolved origin (e.g. from database queries, configuration files, etc) that are subsequently used in SQL queries due to the high likelihood of false positives.

PHP-fusion is different from the other five code bases because it does not directly access HTTP form data from input hash tables such as $_GET and $_POST. Instead it uses the extract operation to automatically import such information into the current variable scope. We describe our findings for PHP-fusion in the following subsection.

4.1 Case Study: Two Exploitable SQL Injection Attacks in PHP-fusion

In this section, we show two case studies of exploitable SQL injection vulnerabilities in PHP-fusion detected by our tool. PHP-fusion is an open-source content management system (CMS) built on PHP and MySQL. Excluding locale specific customization modules, it consists of over 16,000 lines of PHP code and has a wide user-base because of its speed, customizability and rich features. Browsing through the code, it is obvious that the author programmed with security in mind and has taken extra care in sanitizing input before use in query strings.

Our experiments were conducted on the then latest 6.00.204 version of the software. Unlike other code bases we have examined, PHP-fusion uses the extract operation to import user input into the current scope. As an example, extract($_POST, EXTR_OVERWRITE) has the effect of introducing one variable for each key in the $_POST hash table into the current scope, and assigning the value of $_POST[key] to that variable. This feature reduces typing, but introduces confusion for the checker and security vulnerabilities into the software—both of the exploits we constructed involve use of uninitialized variables whose values can be manipulated by the user because of the extract operation.

Since PHP-fusion does not directly read user input from input hashes such as $_GETor $_POST, there are no direct error messages generated by our tool. Instead we inspect warnings (recall the discussion about errors and warnings above), which correspond to security sensitive variables whose definition is unresolved by the checker (e.g., introduced via the extract operation, or read from configuration files).

We ran our checker on all top level scripts in PHP-fusion. The tool generated 22 unique warnings, a majority of which relate to configuration variables that are used in the construction of a large number of queries.[9] After filtering those out, 7 warnings in 4 different files remain.

We believe all but one of the 7 warnings may result in exploitable security vulnerabilities. The lone false positive arises from an unanticipated sanitization:

```
/* php-files/lostpassword.php */
if (!preg_match("/^[0-9a-z]{32}$/", $account))
        $error = 1;
if (!$error) { /* database access using $account */ }
if ($error) redirect("index.php");
```

Instead of terminating the program immediately based on the result from preg_match, the program sets the $error flag to true and delays error handling, which is in general not a good practice. This idiom can be handled by adding slightly more information in the block summary.

We investigated the first two of the remaining warnings for potential exploits and confirmed that both are indeed exploitable on a test installation. Unsurprisingly

[8] Information about the results, along with the source codebases, are available online at:
`http://glide.stanford.edu/yichen/research/`.

[9] Database configuration variables such as $db_prefix accounted for 3 false positives, and information derived from the database queries and configuration settings (e.g. locale settings) caused the remaining 12.

both errors are made possible because of the extract operation. We explain these two errors in detail below.

1) Vulnerability in script for recovering lost password. This is a remotely exploitable vulnerability that allows any registered user to elevate his privileges via a carefully constructed URL. We show the relevant code below:

```
1 /* php-files/lostpassword.php */
2 for ($i=0;$i<=7;$i++)
3     $new_pass .= chr(rand(97, 122));
4 ...
5 $result = dbquery("UPDATE ".$db_prefix."users
6   SET user_password=md5('$new_pass')
7   WHERE user_id='".$data['user_id']."'");
```

Our tool issued a warning for $new_pass, which is uninitialized on entry and thus defaults to the empty string during normal execution. The script proceeds to add seven randomly generated letters to $new_pass (lines 2-3), and uses that as the new password for the user (lines 5-7). The SQL request under normal execution takes the following form:

```
UPDATE users SET user_password=md5('???????')
  WHERE user_id='userid'
```

However, a malicious user can simply add a new_pass field to his HTTP request by appending, for example, the following string to the URL for the password reminder site:

&new_pass=abc%27%29%2cuser_level=%27103%27%2cuser_aim=%28%27

The extract operation described above will magically introduce $new_pass in the current variable scope with the following initial value:

abc'), user_level =' 103', user_aim = ('

The SQL request is now constructed as:

```
UPDATE users SET user_password=md5('abc'),
      user_level='103', user_aim=('???????')
  WHERE user_id='userid'
```

Here the password is set to "abc", and the user privilege is elevated to 103, which means "Super Administrator." The newly promoted user is now free to manipulate any content on the website.

2) Vulnerability in the messaging sub-system. This vulnerability exploits another use of potentially uninitialized variable $result_where_message_id in the messaging sub system. We show the relevant code in Figure 5.

```
1  if (isset($msg_view)) {
2    if (!isNum($msg_view)) fallback("messages.php");
3    $result_where_message_id="message_id=".$msg_view;
4  } elseif (isset($msg_reply)) {
5    if (!isNum($msg_reply)) fallback("messages.php");
6    $result_where_message_id="message_id=".$msg_reply;
7  }
8  ... /* ~100 lines later */ ...
9  } elseif (isset($_POST['btn_delete']) ||
10   isset($msg_delete)) { // delete message
11   $result = dbquery("DELETE FROM ".$db_prefix.
12    "messages WHERE ".$result_where_message_id. // BUG
13    " AND ".$result_where_message_to);
```

Figure 5: An exploitable vulnerability in PHP-fusion 6.00.204.

Our tool warns about unsanitized use of $result_where_message_id. On normal input, the program initializes $result_where_message_id using a cascading `if` statement. As shown in the code, the author is very careful about sanitizing values that are used to construct $result_where_message_id. However, the cascading sequence of `if` statements does not have a default branch. And therefore, $result_where_message_id might be uninitialized on malformed input. We exploit this fact, and append

&request_where_message_id=1=1/*

The query string submitted on line 11-13 thus becomes:

```
DELETE FROM messages WHERE 1=1 /* AND ...
```

Whatever follows "/*" is treated as comments in MySQL and thus ignored. The result is loss of all private messages in the system. Due to the complex control and data flow, this error is unlikely to be discovered via code review or testing.

We reported both exploits to the author of PHP-fusion, who immediately fixed these vulnerabilities and released a new version of the software.

5 Related Work

5.1 Static techniques

WebSSARI is a type-based analyzer for PHP [7]. It uses a simple intraprocedural tainting analysis to find cases where user controlled values flow into functions that require trusted input (i.e. *sensitive functions*). The analysis relies on three user written "prelude" files to provide information regarding: 1) the set of all sensitive functions–those require sanitized input; 2) the set of all untainting operations; and 3) the set of untrusted input variables. Incomplete specification results in both substantial numbers of false positives and false negatives.

WebSSARI has several key limitations that restrict the precision and analysis power of the tool:

1. WebSSARI uses an intraprocedural algorithm and thus only models information flow that does not cross function boundaries.

   Large PHP codebases typically define a number of application specific subroutines handling common operations (e.g., query string construction, authentication, sanitization, etc) using a small number of system library functions (e.g., mysql_query). Our algorithm is able to automatically infer information flow and pre- and post-conditions for such user-defined functions whereas WebSSARI relies on the user to specify the constraints of each, a significant burden that needs to be repeated for each source codebase examined. Examples in Section 3.3 represent some common forms of user-defined functions that WebSSARI is not able to model without annotations.

To show how much interprocedural analysis improves the accuracy of our analysis, we turned off function summaries and repeated our experiment on `News Pro`, the smallest of the five codebases. This time, the analysis generated 19 error messages (as opposed to 8 with interprocedural analysis). Upon inspection, all 11 extra reports are false positives due to user-defined sanitization operations.

2. WebSSARI does not seem to model conditional branches, which represent one of the most common forms of sanitization in the scripts we have analyzed. For example, we believe it will report a false warning on the following code:

   ```
   if (!is_numeric($_GET['x']))
       exit;
   mysql_query(“... $_GET['x'] ...'');
   ```

   Furthermore, interprocedural conditional sanitization (see the example in Section 3.1.6) is also fairly common in codebases.

3. WebSSARI uses an algorithm based on static types that does not specifically model dynamic features in scripts. For example, dynamic typing may introduce subtle errors that WebSSARI misses. The include statement, used extensively in PHP scripts, dynamically inserts code to the program which may contain, induce, or prevent errors.

We are unable to directly compare the experimental results due to the fact that neither the bug reports nor the WebSSARI tool are available publicly. Nor are we able to compare false positive rates since WebSSARI reports per-file statistics which may underestimate the false positive ratio. A file with 100 false positives and 1 real bug is considered to be “vulnerable” and therefore does not contribute to the false positive rate computed in [7].

Livshits and Lam [9] develop a static detector for security vulnerabilities (e.g., SQL injection, cross site scripting, etc) in Java applications. The algorithm uses a BDD-based context-sensitive pointer analysis [19] to find potential flow from untrusted sources (e.g., user input) to trusting sinks (e.g., SQL queries). One limitation of this analysis is that it does not model control flow in the program and therefore may misflag sanitized input that subsequently flows into SQL queries. Sanitization with conditional branching is common in PHP programs, so techniques that ignore control flow are likely to cause large numbers of false positives on such code bases.

Other tainting analysis that are proven effective on C code include CQual [4], MECA [21], and MC [6, 2]. Collectively they have found hundreds of previously unknown security errors in the Linux kernel.

Christensen *et. al.* [3] develop a string analysis that approximates string values in a Java program using a context free grammar. The result is widened into a regular language and checked against a specification of expected output to determine syntactic correctness. However, syntactic correctness does not entail safety, and therefore it is unclear how to adapt this work to the detection of SQL injection vulnerabilities. Minamide [10] extends the approach and construct a string analyzer for PHP, citing SQL injection detection as a possible application. However, the analyzer models a small set of string operations in PHP (e.g., concatenation, string matching and replacement) and ignores more complex features such as dynamic typing, casting, and predicates. Furthermore, the framework only seems to model sanitization with string replacement, which represents a small subset of all sanitization in real code. Therefore, accurately pinpointing injection attacks remains challenging.

Gould *et. al.* [5] combines string analysis with type checking to ensure not only syntactic correctness but also type correctness for SQL queries constructed by Java programs. However, type correctness does not imply safety, which is the focus of our analysis.

5.2 Dynamic Techniques

Scott and Sharp [15] propose an application-level firewall to centralize sanitization of client input. Firewall products are also commercially available from companies such as NetContinuum, Imperva, Watchfire, etc. Some of these firewalls detect and guard against previously known attack patterns, while others maintain a white list of valid inputs. The main limitation here is that the former is susceptible to both false positives and false negatives, and the latter is reliant on correct specifications, which are difficult to come by.

The Perl taint mode [12] enables a set of special security checks during execution in an unsafe environment. It prevents the use of untrusted data (e.g., all command line arguments, environment variables, data read from files, etc) in operations that require trusted input (e.g., any command that invokes a sub-shell). Nguyen-Tuong [11] proposes a taint mode for PHP, which, unlike the Perl taint mode, not define sanitizing operations. Instead, it tracks each character in the user input individually, and employs a set of heuristics to determine whether a query is safe when it contains fragments of user input. For example, among others, it detects an injection if an operator symbol (e.g., “(”, “)”, “%”, etc) is marked as tainted. This approach is susceptible to both false positives and false negatives. Note that static analyses are also susceptible to both false positives and false negatives. The key distinction is that in static analyses, inaccuracies are resolved at compile time instead of at runtime, which is much less forgiving.

6 Conclusion

We have presented a static analysis algorithm for detecting security vulnerabilities in PHP. Our analysis employs a novel three-tier architecture that enables us to handle dynamic features unique to scripting languages such as dynamic typing and code inclusion. We demonstrate the effectiveness of our approach by running our tool on six popular open source PHP code bases and finding 105 previously unknown security vulnerabilities, most of which we believe are remotely exploitable.

Acknowledgement

This research is supported in part by NSF grants SA4899-10808PG, CCF-0430378, and an IBM Ph.D. fellowship. We would like to thank our shepherd Andrew Myers and the anonymous reviewers for their helpful comments and feedback.

References

[1] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, 1994.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *2002 IEEE Symposium on Security and Privacy*, 2002.

[3] A. Christensen, A. Moller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th Static Analysis Symposium*, 2003.

[4] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

[5] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[6] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[7] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, 2004.

[8] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the web, `http://caml.inria.fr`.

[9] V. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[10] Y. Minamide. Approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, 2005.

[11] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th International Information Security Conference*, 2005.

[12] Perl documentation: Perlsec. `http://search.cpan.org/dist/perl/pod/perlsec.pod`.

[13] PHP: Hypertext Preprocessor. `http://www.php.net`.

[14] PHP usage statistics. `http://www.php.net/usage.php`.

[15] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[16] Security space apache module survey (Oct 2005). `http://www.securityspace.com/s_survey/data/man.200510/apachemods.html`.

[17] Symantec Internet security threat report: Vol. VII. Technical report, Symantec Inc., Mar. 2005.

[18] TIOBE programming community index for November 2005. `http://www.tiobe.com/tpci.htm`.

[19] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[20] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst.*, 19(1):87–152, Jan. 1997.

[21] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th Conference on Computer and Communications Security*, 2003.