SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications

Sooel Son
The University of Texas at Austin
samuel@cs.utexas.edu

Vitaly Shmatikov
The University of Texas at Austin
shmat@cs.utexas.edu

Abstract

Web applications are vulnerable to semantic attacks such as denial of service due to infinite loops caused by malicious inputs and unauthorized database operations due to missing security checks. Unlike "conventional" threats such as SQL injection and cross-site scripting, these attacks exploit bugs in the logic of the vulnerable application and cannot be discovered using data-flow analysis alone.

We give the first characterization of these types of vulnerabilities in PHP applications, develop novel inter-procedural algorithms for discovering them in PHP source code, and implement these algorithms as part of SaferPHP, a framework for static security analysis of PHP applications. SaferPHP uncovered multiple, previously unreported vulnerabilities in several popular Web applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

Keywords Static analysis, data flow analysis, denial of service, security checks, access control, PHP

1. Introduction

Web applications implemented in scripting languages such as PHP are notoriously insecure. A recent report found security issues in 64% of the analyzed websites, with an average of 6 issues per website [32]. Classic Web-application vulnerabilities include SQL injection, cross-site scripting, SSI injection, and cross-site request forgery [1, 3, 22, 29]. These attacks involve injection of malicious commands into Web content presented by the vulnerable application, queries submitted by the application to a back-end database, or requests sent by the browser to the application.

In this paper, we focus on entirely different classes of vulnerabilities which are caused by semantic bugs in the application's logic and—unlike injection vulnerabilities—cannot be characterized simply in terms of undesirable data flows. The first class of vulnerabilities is denial of service due to infinite loops triggered by malicious network inputs. These bugs enable a single packet to crash the server or cause extraordinary resource consumption. The second class is missing authorization checks prior to sensitive database operations. PHP applications are especially prone to missing-check vulnerabilities because they typically consist of multiple files. Each file can be invoked directly from the network by supplying its name as part of the URL and thus represents an unintended entry point, potentially introducing an unchecked path to the sensitive operation.

This paper describes the design and implementation of SaferPHP, a new tool for static security analysis of PHP applications. SaferPHP is the first tool capable of automatically inferring the PHP application's authorization policy and finding missing authorization checks. The problem of identifying security checks in PHP code is significantly more difficult than in, say, Java libraries or SELinux because PHP checks are application-specific and cannot be recognized syntactically; instead, SaferPHP recognizes them by the semantic role they play in the application. Furthermore, SaferPHP is the first security analysis tool to support object-oriented features of PHP.

At the core of SaferPHP are several new static analysis algorithms. To find denial-of-service vulnerabilities, SaferPHP first uses standard taint analysis to find loops whose execution can be influenced by network inputs.[1] It then employs new algorithms based on symbolic execution to determine if the attacker can prevent the loop from terminating, thus causing denial of service. Taint analysis also helps find calls to vulnerable library functions, misuses of uninitialized variables, and SQL injection sites. To find

PLAS'11 June 5, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0830-4/11/06…$10.00

[1] We emphasize that taint analysis *per se* is not enough, and standard taint-based techniques for finding cross-site scripting and SQL injection sites cannot be used for finding DoS vulnerabilities.

missing authorization checks, SaferPHP uses several new inter-procedural algorithms to analyze all calling contexts in which sensitive database operations may be invoked, applies semantic analysis of control dependencies to identify possible security checks, and verifies whether checks are consistently present in all calling contexts. SaferPHP found 49 previously unreported, remotely exploitable vulnerabilities in popular PHP applications (including CVE-2010-2423 and CVE-2010-2424), with very few false positives. These vulnerabilities were reported to the developers and we worked with several of them to patch the vulnerable code.

2. Related work

Prior work on static security analysis of PHP applications focused primarily on data-flow (taint) analysis to find cross-site scripting and SQL injection vulnerabilities, which are characterized by unsanitized inputs flowing to the application's HTML output or back-end SQL queries [18–20, 24, 31, 33]. Taint analysis alone is not sufficient for finding either missing authorization checks, or denial-of-service (DoS) vulnerabilities, which often involve tainted inputs controlling execution of some part of the program (in this paper, we focus on loops whose termination depends on a tainted input). There are no "sinks" to serve as the targets of taint analysis, nor is it enough to find program locations that are control-dependent on tainted data because flagging each one as a potential vulnerability would produce an overwhelming number of false positives.

In this paper, we (1) characterize two new classes of semantic vulnerabilities—DOS due to attacker-controlled infinite loops and missing authorization checks—and (2) present entirely new algorithms for statically finding these vulnerabilities. The only component borrowed from previous literature is taint analysis, which is the preliminary step before the main security analyses.

Static detection of DoS vulnerabilities and termination analysis. Previous research on finding DoS vulnerabilities focused on servers implemented in C. C code can be crashed by a single buffer overflow (*e.g.*, "inputs of death" [7, 21]). Denial of service can also be caused by algorithmic complexity attacks which exploit the worst-case behavior of server data structures [10].

Safer is a tool for finding recursive function calls in C code where the depth of recursion is superlinear in the size of the input [8]. By contrast, we focus on attacker-controlled infinite loops. This is a very different problem. Except for the preliminary taint analysis, our algorithms are new and completely different from [8].

Non-terminating loops can be found by runtime analysis [6], but only after the program has already entered the loop. By contrast, SaferPHP is a static analysis tool. Proving loop termination, while undecidable in general, is a subject of active research [5, 9]. Our techniques are somewhat similar to those used for proving loop *non*-termination [17], but the approach used in [17] is not suitable for finding DoS vulnerabilities because it depends on *dynamically* finding candidate loops. In our experience, DoS vulnerabilities tend to occur on rarely executed paths and are unlikely to manifest in normal execution. By contrast, SaferPHP *statically* finds all loops whose termination is controlled by the attacker and attempts to demonstrate the existence of inputs that cause the loop not to terminate.

Finding missing authorization checks. Prior work on finding missing authorization checks without a specification focused on Java [23, 28] and SELinux [14, 30]. All of these analyses assume that authorization checks are syntactically recognizable, *e.g.*, as invocations of the Java security manager [23, 28] or calls to functions such as `security_file_permission` and `security_file_mmap` in SELinux [30]. Security invariants in Java can also be learned and enforced using dynamic analysis [4, 15].

Verifying the presence of known security functions in the call chain prior to every sensitive operation—as done in [14, 23, 28, 30]—does not work for PHP applications. In general, PHP authorization checks are not recognizable syntactically because PHP does not have a standard set of security functions (see Section 4.6). SaferPHP is the first system to use consistency analysis of control dependencies in the whole-program control-flow graph to identify authorization checks *semantically*, by their role in the program, without any specification or syntactic characterization of the application's authorization policy.

Balzarotti *et al.* proposed an approach for finding unintended work flows caused by multiple PHP modules [2]. This analysis can find some missing checks but, in general, is too coarse-grained: for example, a missing check on one of the many execution paths leading to a sensitive operation within the same module is not a multi-module workflow violation according to the definition of [2]. Therefore, many of the vulnerabilities discovered by SaferPHP could not have been found using the tool described in [2].

3. Example vulnerabilities

The current prototype of SaferPHP can discover five types of vulnerabilities: denial of service (DoS) due to attacker-controlled infinite loops, unauthorized access to sensitive operations, SQL injection, misuse of uninitialized variables, and DoS due to vulnerable PHP library functions. Example 1 illustrates all five types with simplified examples.

In a typical PHP application, authorization is usually required prior to to database, cookie, and file operations. In the current prototype of SaferPHP, we define sensitive operations to be all database operations other than `select`, but extending it to other operations presents no conceptual difficulties. Many `selects` (*e.g.*, those responsible for retrieving credentials from the database when authenticating a user or deciding if the IP address of an incoming request is blacklisted) can be legitimately reached without any check. In the

Example 1 Illustrations of PHP vulnerabilities

```
Ln1:  // No access check such as "if(!$logged_in) die;"
Ln2:  if (isset($_GET[’post_id’]))
          $post_id = $_GET[’post_id’];
Ln3:  $sql="DELETE FROM blogdata WHERE post_id=$post_id";
Ln4:  $query=mysql_query($sql)
          or die("Cannot query the database.<br>");
Ln5:  ...
Ln6:  if(isset($varUninitialized))
         echo($varUninitialized);
Ln7:  ...
Ln8:  if (isset($_GET[’content’]))$str=$_GET[’content’];
Ln9:  if (isset($_GET[’eol’]))$eol=$_GET[’eol’];
Ln10: $encoded=chunk_split($str,76,$eol);
Ln11: $value=unserialize(stripslashes($_POST[$afield]));
Ln12: ...
Ln13: if(isset($_GET[’year’])) $year = $_GET[’year’];
Ln14: $i = 1962;
Ln15: while( $i<=$year )
Ln16: {
Ln17:    if( $i < 3000 ){ processYear($i); }
Ln18:    else { $i=$year; continue; }
Ln19:    $i++;
Ln20: }
```

absence of a specification denoting which operations must be protected, treating `select` as sensitive would have generated an overwhelming number of false positives.

If Line 1 is the program entry point, there is no authorization check prior to the database operation on Line 3. By typing the URL of this PHP file directly into the browser's address bar, a malicious user can execute a sensitive SQL operation (deleting from the database). Note that this is *not* an instance of SQL injection.

Lines 13 to 20 are an example of a DoS vulnerability. By controlling the value of $year, the adversary can cause the loop to be executed any number of times. If $year $>$ 3000, the loop becomes infinite, leading to excessive CPU consumption and denial of service to legitimate users.

Lines 2 to 4 show an example of a SQL injection vulnerability. Here, unsanitized user input *post_id* is used as part of a SQL query, enabling the attacker to change the structure of the query and execute an arbitrary SQL statement. See [1] for a detailed explanation of SQL injection attacks.

In PHP applications, the initial assignment and uses of a variable may occur in different files. If the attacker directly executes the PHP file containing only the uses, he may supply his own initial value via a URL variable if "register_global" is set (see Section 5.3). Consider Line 6 of Example 1. Even though *$varUninitialized* is not intended to hold user inputs, the attacker may supply an initial value containing a malicious script, leading to cross-site scripting.

Lines 8 to 11 show two DoS vulnerabilities due to calls to vulnerable library functions. Line 10 causes the PHP interpreter whose version is lower than 5.2.3 to crash if the lengths of $str and $eol are over 65534 [11]. Line 11 with a specially crafted input will cause resource exhaustion in PHP whose version is lower than 5.3.0 [12].

4. Static analysis framework

After parsing PHP source code, SAFERPHP builds the whole-program call graph and control-flow graph (CFG), then uses them to collect critical variables which control reachability of sensitive database operations and to perform inter-procedural, context-sensitive taint analysis. Information about critical variables is used to find contexts in which sensitive database operations may be executed without authorization checks. Taint analysis is used to find loops whose termination depends on external inputs; SAFERPHP uses symbolic execution to determine if the attacker can cause a loop not to terminate. Taint analysis is only a preliminary step to finding DoS vulnerabilities, but it also helps discover "conventional" vulnerabilities caused by unsanitized inputs.

4.1 Architecture of SAFERPHP

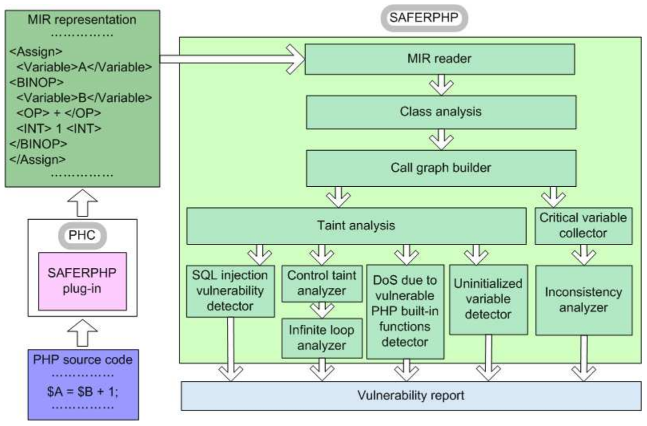
Our SAFERPHP prototype is based on an open-source PHP compiler called PHC [26]. SAFERPHP employs only the PHC front-end, which parses the source code and produces AST (Abstract Syntax Tree), HIR, and MIR (respectively, High and Medium Intermediate Representation). PHC provides APIs for plugins, which access the intermediate representation of a parsed PHP program by traversing the syntax tree and possibly modifying its nodes. We implemented a plugin to re-format MIR for our purposes.

The modular structure of SAFERPHP is shown in Fig. 1. The basic information about the program (CFG, call graph, taint relations, *etc.*) is used by many different analyses. A separate module is responsible for each type of analysis and may depend on other modules. Security analysis of a PHP program starts with reading its MIR and ends with generating a summary of potential vulnerabilities. Information computed by the basic static analysis modules is fed into modules responsible for finding security vulnerabilities. For example, call-graph information is used by the taint-analysis module, whose output in turn is used to find SQL injection vulnerabilities, potential infinite loops, and misuses of uninitialized variables. This design makes it easy to develop additional analysis modules for other kinds of vulnerabilities.

4.2 Computing the control-flow graph

In SAFERPHP, the basic unit of control-flow analysis is a function. Each function declaration is associated with one control-flow graph (CFG). A set of control blocks which does not belong to any function is the CFG of the main function of a given file. When SAFERPHP encounters a class declaration, it builds an internal data structure whose identifier is the name of the class. It contains the set of declared field variables and pointers to the CFGs of member functions. A concatenation of the file (resp., class) name and function name uniquely identifies the CFG of a function (resp., member function) in the global namespace.

A typical PHP application consists of multiple files. It is common practice in PHP programming to apply a merging



```
..............
<Assign>
  <Variable>A</Variable>
<BINOP>
  <Variable>B</Variable>
  <OP> + </OP>
  <INT> 1 <INT>
</BINOP>
</Assign>
..............
```

```
..............
$A = $B + 1;
..............
```

Figure 1: Architecture of SAFERPHP.

function such as $include$ or $require$ to reference another file. For accurate analysis, it is essential to construct the whole-program CFG by taking these functions into account.

It can be difficult to determine the arguments of merging functions statically because of user inputs and undecidable expressions. SAFERPHP collects partial information which explicitly appears in the arguments and matches it to file names. If there is only one match between statically possible arguments and names, SAFERPHP merges the identified file into the main program. If there are multiple possible matches or no match, SAFERPHP asks the user which file is meant to be included at the given point. After collecting reference relations among source files, SAFERPHP uses the $include$ option of PHC to merge files into one MIR. The subsequent analyses are based on this merged representation.

4.3 Computing the call graph

None of the previous tools for static security analysis of PHP code handle object-oriented features of PHP. There exist standard algorithms for building call graphs of object-oriented programs [13, 16], but PHP presents challenges.

First, destinations of function calls may change at runtime. For example, $\$foo()$ is an indirect call, in which the called function may be changed during program execution by changing the value of the string variable $\$foo$. Assignment of callback functions and function renaming further complicate static call-graph construction.

Second, PHP is not statically typed. The type of a variable can be changed dynamically by assigning to it a value of a different type or calling a function. Consider a statement $\$A->foo()$. To build the corresponding edge in the call graph, it is necessary to determine statically the class whose instance will be stored in variable $A at runtime. This can be done based just on the function name, but, in general, $foo()$ may exist in several classes, or parent and children classes may provide different implementations of $foo()$.

When building the whole-program CFG, SAFERPHP only considers direct function calls, in which the name of the called function appears explicitly in the source code and MIR. For non-member functions, SAFERPHP creates an edge in the whole-program CFG from the callsite to the entry point of the function's CFG. If the function is a member function, additional analysis is needed due to the lack of static information about the type of the owner variable.

SAFERPHP distinguishes class instances when analyzing accesses to field variables of objects, but not when analyzing calls to member functions. For each occurrence of the `new` operator in the MIR, SAFERPHP creates a data structure representing an instance. It has separate reserved space for field variables and a list of pointers to the CFGs of member functions. For example, consider these statements: $\$A = \text{new } classA(), \$B = \text{new } classA()$. The fields of $A and $B refer to different variables even though they have the same name. The member functions of both instances, however, refer to the same CFGs.

After generating class instances, SAFERPHP propagates them within each CFG through assignment statements. This intra-procedural class analysis enables SAFERPHP to track which instances are stored in which variable. To track instances corresponding to "this" keyword, SAFERPHP also stores instances associated with the owner variable for each invocation of a member function.

Static class analysis is conservative, thus a variable may be associated with a *set* of instances. SAFERPHP first checks whether the member function being called is the same for all instances associated with the variable. If so, only one function may be called at this callsite, regardless of which instance is stored in the variable, and one call-graph edge is sufficient. Otherwise, SAFERPHP chooses the first statically feasible edge. We deliberately opted for this (generally unsound) approach instead of supporting multiple CFG edges from a single callsite, which would have been more precise at the cost of significantly increasing complexity of the analysis. In our experiments (see Section 6), even applications that utilize object-oriented features of PHP do not contain any callsites with more than one callee. Adding support for non-deterministic CFGs would improve theoretical soundness, but is unlikely to help discover any new security vulnerabilities of practical significance.

If the destination of a call-graph edge is an unidentified function (*i.e.*, it was not encountered in the first phase of the analysis), we assume that it is a built-in PHP function. A user-defined function may be misclassified as a built-in function. In our experiments, fewer than 3% of method invocations could not be resolved; most were caused by plugin methods with missing bodies.

4.4 Taint analysis

The purpose of taint analysis is to determine whether a given variable at a given point in the program contains a value which came from an untrusted source and may be controlled by the attacker. Formally, x is tainted at node n in a function f under context c if and only if there is at least one path from the root that reaches n under c and the value of x is data-dependent on some user input; x is data-dependent on y if and only if there exists a reaching definition of the form $x = e$ such that e is an expression containing y or another variable z which is data-dependent on y [8]. In SAFERPHP, a user input is the return value of any function $q \in Q$ through which external inputs enter the program, or a global array variable $s \in G$ pre-defined by SAFERPHP.

Term definitions

n, m : nodes in a CFG $\quad n \xrightarrow{stat} m$: transition from n to m via $stat$ at n

$T(c, f, n, x)$: var x is tainted at node n in procedure f under calling context c

Taint relation propagation rules

$$\frac{n \xrightarrow{x:=y} m, T(c,f,n,y)}{T(c,f,m,x)} \qquad \frac{n \xrightarrow{x:=\text{a binop b}} m, \text{T(c,f,n,a or b)}}{T(c,f,m,x)}$$

$$\frac{n \xrightarrow{x:=y[any]} m, T(c,f,n,y)}{T(c,f,m,x)} \qquad \frac{n \xrightarrow{x[any]:=y} m, T(c,f,n,y)}{T(c,f,m,x)}$$

$$\frac{n \xrightarrow{x \rightarrow a:=y} m, T(c,f,n,y)}{T(c,f,m,\text{field a of x})} \qquad \frac{n \xrightarrow{x:=y \rightarrow a} m, T(c,f,n,\text{field a of y})}{T(c,f,m,x)}$$

$$\frac{n \xrightarrow{x:=r(y1,\ldots,y_i,\ldots y_k)} m, r \notin P_{identified}, T(c,f,n,y_i))}{T(c,f,m,x)}$$

$$\frac{n \xrightarrow{x:=r(y1,\ldots,y_i,\ldots y_k)} m, r \in P_{identified}, T(c,f,n,y_i)}{T(r :: c, f, entry(r), arg_i)}$$

$$\frac{n \xrightarrow{x:=r(y)} m, r \in P, T(r :: c, f, exit(r), retVal(r))}{T(c,f,m,x)}$$

Figure 2: Taint analysis rules.

SAFERPHP treats all files and direct user inputs as tainted. Pre-defined taint sources are the global array variables $G = \{\_GET, \_POST, \_COOKIE, \_REQUEST\}$, which typically contain unsanitized network inputs, and the return values of external functions dealing with files: $Q = \{fgets, file\_get\_contents, readfile, readlink, \ldots\}$. Database records are not considered tainted; it is common for Web applications to operate on databases and tainting the entire database would effectively taint all data values. Therefore, SAFERPHP may miss attacks due to second-order taints, *i.e.*, if the database contains malicious values stored by another application or directly by the attacker.

Taint analysis in SAFERPHP is inter-procedural and context-sensitive. The call graph includes identified user-defined functions (Section 4.3). For unidentified functions ($r \notin P_{identified}$), SAFERPHP assumes that their return values are tainted if any argument is tainted. To propagate taint information, SAFERPHP computes the taint relation $T(c, f, n, x)$, denoting that variable x is tainted at node n in procedure f under the calling context c, using the rules of Fig. 2. T propagates through assignments, with special rules for arrays and field variables. If a tainted variable is assigned to an array variable, SAFERPHP taints the entire array. If a tainted variable is assigned to a field of an owner variable, SAFERPHP taints the corresponding field of every possible instance statically associated with this owner variable. To compute the taint relation for the entire program, the rules of Fig. 2 are applied until a fixed point is reached.

Our taint analysis employs intra-procedural alias analysis (this is sufficient for our purposes). Each variable is associated with an alias list. If a variable appears as the target of an assignment containing &, the source variable is added to its alias list and vice versa. For example, the `$a =& $b` statement results in adding $b to the alias list of $a and $a to the alias list of $b. If some taint inference rule marks $a (resp., $b) as tainted, SAFERPHP consults the alias list and also marks $b (resp., $a) as tainted.

4.5 Tainted loops and symbolic execution

To find denial-of-service vulnerabilities, SAFERPHP must first find loops whose termination is potentially controlled by the attacker via tainted variables. A loop is a set of nodes that has at least one back edge [25], *i.e.*, an edge whose destination node dominates the source node. To compute all loops in a CFG, SAFERPHP applies an analysis module which, given a source, produces its dominators.

Algorithm 1 Deciding whether a loop is tainted

```
L: set of all nodes in one loop
E: set of CFG edges
p: procedure
TL: set of detected tainted loops TL
TS(d,p): true if node d is a branch and any variable used at node d is tainted

function computeTL( p, L )
{
  for $s \in L$ do
    for $(s, c) \in E$ do
      if $c \notin L$ then
        for $d \in \{$ n | n dominate s and $n \in L\}$ do
          if TS(d,p) then
            TL ← TL $\cup \{L\}$;
            break;
}
```

Algorithm 1 checks whether a loop is tainted. Assuming there exists an edge exiting loop L, if the source node of the exiting edge is dominated by a branch node n and there exists a tainted variable x which controls the branching condition in n, then loop L is tainted.

Flagging every tainted loop as a potential vulnerability would produce a prohibitive number of false positives. To find real vulnerabilities, SAFERPHP implements a new algorithm: it performs intra-procedural symbolic execution of all one-iteration paths within tainted loops, looking for paths where the variables involved in the exit condition do not change. This heuristic does not scale to the entire CFG, but is feasible when applied only to tainted loops.

Symbols. SAFERPHP uses 7 symbols to represent variable values during symbolic execution. SymbolInt represents an integer constant and carries the same during symbolic exe-

Type of equivalence	Equivalence condition
SymbolInt A ≡ SymbolInt B	Value of A = Value of B
SymbolRangeInt A ≡ SymbolInt B	Min value of A $\leq$ Value of B $\leq$ Max value of A
SymbolRangeInt A ≡ SymbolRangeInt B	There is an overlap between the ranges of A and B
SymbolString A ≡ SymbolString B	Value of A = Value of B
SymbolBool A ≡ SymbolBool B	Bool value of A = Bool value of B or Bool value of A or B is unknown
SymbolArbitraryInt A ≡ SymbolArbitraryInt B	Variation of A = Variation of B and Variable name of A = Variable name of B
SymbolArbitrary A ≡ SymbolArbitrary B	Update counter of A = Update counter of B and Variable name of A = Variable name of B

Table 1: Equivalence rules between symbols.

cution. SymbolString represents a string. SymbolBool represents a Boolean; its value is True, False, or Unknown. SymbolRangeInt represents any integer value whose minimum and maximum values are statically known. SymbolArbitraryInt represents an arithmetic change in the value of a variable. For example, after the `$i = $i+2` statement, SymbolArbitraryInt holds the name of the variable (`$i`) and the change in its value ($+2$). A variable may be associated with multiple symbols unless it's a SymbolArbitrary.

SymbolArbitraryString represents any string whose minimum and maximum lengths are statically known. It carries these lengths and the set of substrings contained in the string. For example, after the statement `$str = substr($str, 0, 3)`, `str` is either a Boolean `False`, or a string whose length is between 0 and 3.

SymbolArbitrary represents a statically unknown value. It carries the variable name and the update counter. During symbolic execution, any statement which may modify this variable increments the counter. For example, consider the statement `$c = functionA($c)`. If `c` is represented by a SymbolArbitrary, then its update counter is incremented. If `c` is not associated with any symbol, SAFERPHP assigns to it a SymbolArbitrary symbol with the counter of 0.

Equivalence of symbols. To find attacker-controlled infinite loops, SAFERPHP must decide if certain variables may be equal at runtime by considering the symbols representing their values during symbolic execution. The rules for symbol equivalence (shown in Table 1) are necessarily conservative, *i.e.*, SAFERPHP determines if two variables *may* hold the same value at runtime. This may lead to false positives. For example, given two SymbolBools whose values are statically unknown, SAFERPHP will conservatively decide that they are equivalent. If two symbols do not satisfy any of the conditions in Table 1, they are assumed to be not equivalent.

Symbolic transition rules. SAFERPHP includes a small interpreter which takes as inputs a PHP statement and symbolic values of variables involved in it. The interpreter employs 20 symbolic transition rules, representing the effect of binary operators such as +, -, *, /,% and comparison operators. For example, consider the statement `$a = $b + 2`. If there is only a SymbolInt symbol associated with variable `b`, then the transition rule associates a SymbolInt with variable `a`. If the operation in the statement does not match any of the transition rules, then SAFERPHP associates a SymbolArbitrary symbol with the target variable of the assignment. If the target variable is also a SymbolArbitrary, SAFERPHP increments its update counter.

Symbolic execution of built-in PHP functions such as `substr` and `strlen` requires more precision. We implemented symbolic transition rules for the ten most common built-in string processing functions. (Of course, there are many more built-in PHP functions; we leave their symbolic modeling to future work.) If the built-in function is not one of those currently supported by SAFERPHP, its return value is represented as a SymbolArbitrary. We omit the complete list of symbolic transition rules due to space limitations.

4.6 Collecting critical variables

PHP applications usually mediate access to sensitive operations such as database-modifying SQL queries by checking that the user is logged in, authorized to perform the operation, *etc.* Unlike Java [23, 28], PHP has neither a standard protection paradigm, nor a standard set of security-checking functions or variables. The semantics of security checks are highly application-specific. SAFERPHP takes advantage of the following observations: (1) any check involves a branch statement which depends on the value of some *critical variable*, (2) one of the branches corresponds to the failed check and thus results in *not* reaching the sensitive operation, and (3) any context in which the sensitive operation may be executed must contain such a branch statement.

A variable v is *critical* if v or a variable holding a value computed from v appears in a branch statement which affects reachability of a sensitive database operation. Relation $SV(c, v, sp)$ denotes that variable v is critical with respect to a sensitive program location sp under calling context c. Fig. 3 shows the rules for computing $SV(c, v, sp)$. If there is any (possibly inter-procedural) path reaching the exit node from node n without passing through node sp, sp is *control-dependent* on n. $CD(c, n)$ contains every node which is control-dependent on node n under calling context c. $OP(c, v, n)$ denotes the target variable of an assignment where one of the operands is variable v at node n under calling context c. $OP^*(c, v, n)$ denotes the final target variable of the assignment at node n whose origin operand is variable v under calling context c. $ALIAS(v_o, v)$ denotes that two variables are aliases of each other (see Section 4.4).

Not all critical variables are associated with authorization checks. To filter the list of critical variables, SAFERPHP

$$\frac{sp \in CD(c,n),\ n \xrightarrow{x:=OP^*(c,v,n)} m}{SV(c,v,sp)} \qquad \frac{SV(c,v_o,sp),\ ALIAS(v_o,v)}{SV(c,v,sp)}$$

$$\frac{\text{under context c, } n \xrightarrow{x:=y} m}{x = OP(c,y,n)} \qquad \frac{\text{under context c, } n \xrightarrow{x:=\text{a binop b}} m}{x = OP(c,a,n),\ x = OP(c,b,n)}$$

$$\frac{\text{under context c, } n \xrightarrow{x:=y[any]} m}{x = OP(c,y,n)} \qquad \frac{\text{under context c, } n \xrightarrow{x[any]:=y} m}{x = OP(c,y,n)}$$

$$\frac{\text{under context c, } n \xrightarrow{x\rightarrow a:=y} m}{\text{field a of x} = OP(c,y,n)} \qquad \frac{\text{under context c, } n \xrightarrow{x:=y\rightarrow a} m}{x = OP(c,\text{field a of y},n)}$$

$$\frac{\text{under context c, } n \xrightarrow{x:=r(y1,\ldots,y_i,\ldots y_k)} m}{x = OP(c,y_i,n)}$$

Figure 3: Rules for finding critical variables.

relies on the assumption that authorization checks must be consistently performed in *all* calling contexts in which a sensitive operation may be executed (Section 5.2). We stress the novelty of this approach. It relies on inter-procedural analysis of the whole-program control-flow graph and, unlike syntactic approaches such as AutoISES [30], discovers authorization checks by their semantic role in the program.

5. Security analysis modules

5.1 Finding denial-of-service vulnerabilities

To find potential DoS vulnerabilities due to loops whose termination depends on unsanitized network inputs, SaferPHP collects all tainted loops in the program and uses symbolic execution (Section 4.5) to find paths which do not change the variables involved in the loop termination condition, as shown in Algorithm 2.

For each tainted loop, SaferPHP finds the conditional statement one of whose branches is a loop exit and collects all variables which are directly related to the exit condition (*exit-condition variables*). A variable x is "directly related" to y at some program point if and only if there exists a reaching definition of the form $x = e$ such that e is an assignment or a binary operation containing y or some variable z which is directly related to y. After computing the set of exit-condition variables, SaferPHP computes every possible one-iteration *initial path* from the head of the loop to the conditional exit branch, as well as every possible one-iteration *exploring path* from the exit branch back to itself.

For illustration, consider Figure 4. Loop L consists of blocks L1, L2, L3, L4, B2, B5. The exit node is the branch block B5. SaferPHP computes all variables which are directly related to variable $TLE2 in block B5: $i, $TLE2, $TLE1 and $b. The set of initial paths consists of two paths: $L1 \rightarrow B2 \rightarrow L2 \rightarrow L4 \rightarrow B5$ and $L1 \rightarrow B2 \rightarrow L3 \rightarrow L4 \rightarrow B5$. The set of exploring paths also consists of two paths: $B5 \rightarrow L1 \rightarrow B2 \rightarrow L2 \rightarrow L4 \rightarrow B5$ and $B5 \rightarrow L1 \rightarrow B2 \rightarrow L3 \rightarrow L4 \rightarrow B5$.

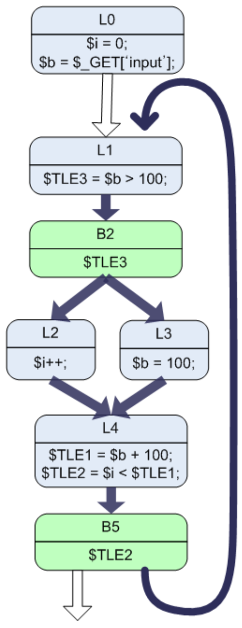


Figure 4: An example of a tainted loop.

SaferPHP then symbolically executes one of the initial paths, stores the resulting symbolic values of the exit-condition variables, and symbolically executes one of the exploring paths. If the values of the exit-condition variables at the beginning and the end of the exploring path are not equivalent (see Section 4.5), SaferPHP tries another path, and so on. Any path in which the symbolic values of the exit-condition variables do not change is reported as a potential vulnerability due to an attacker-controlled infinite loop.

This algorithm is a heuristic (the problem is undecidable in general). It may miss attacks because it considers only a single loop iteration—there may exist an infinite loop in which the value of a loop variable decreases in one iteration and increases in another one (we never observed such a loop in a real-world PHP application). Some reported vulnerabilities may be false positives because our rules for symbol equivalence are conservative. For example, if SaferPHP cannot statically determine anything about the values of some variables, it will declare that they are symbolically equivalent even if their runtime values are never equal. Our experience with SaferPHP shows, however, that this heuristic performs well in practice, finding previously unreported, remotely exploitable denial-of-service vulnerabilities in real-world applications.

5.2 Finding missing authorization checks

Collecting calling contexts of sensitive database operations. SaferPHP uses a predefined list of functions (*e.g.*, `mysql_query`) to find program locations (*sinks*) which execute database operations. For each sink, SaferPHP finds all calling contexts by computing the control-flow and call graphs of all of the application's PHP files, assuming that *any of these files may be used as the program entry point*.

Algorithm 2 Collecting infinite loop candidates

```
function determineLoopType( param LOOP )
{
 set EXIT_COND_VAR = extractExitConditionVar(LOOP);
 if( any variable of EXIT_COND_VAR is not tainted )
     return NOT_TAINT_LOOP;
 set INIT_PATHSET = extractAllInitialPaths( LOOP );
 set PATHSET      = extractAllExploringPaths( LOOP );

 for(each INIT_PATH in INIT_PATHSET) {
  for(every sequential statement STAT in INIT_PATH)
  {  SymbolicExecution( STAT );  }
  set InitialSymbolValueSet=getSymbolStat(EXIT_COND_VAR);
  for(each PATH in PATHSET) {
   for(every sequential statement STAT in PATH)
   {  SymbolicExecution( STAT );  }
   set AfterSymbolValueSet=getSymbolStat(EXIT_COND_VAR);
   bool bInfiniteLoop = true;
   for ( each variable VAR in EXIT_COND_VAR ) {
    if( some value of VAR in InitalSymbolValueSet !=
        some value of VAR in AfterSymbolValueSet)
        bInfiniteLoop = false;
   }
   return bInfiniteLoop;
  }
 }
}
```

When generating call graphs, SAFERPHP records the caller and callee at each callsite. It then uses depth-first search, starting from the sink, to build a tree of calling contexts whose root is the sink and whose leaves are program entry points (even with circular calling contexts, it is enough to record each invoked function once, so the result is always a tree). Fig. 5 shows two trees whose roots are different invocations of the same database operation. For each calling context, SAFERPHP first collects every defined constant and branch node such that the constant appears at the branch node dominating application exit calls and sinks. SAFERPHP then eliminates calling contexts that check undefined constants because such a context cannot reach a sink. Then, SAFERPHP inter-procedurally propagates string constants and determines the type of the database operation at the sink. If the operation is `insert`, `delete`, `update`, or statically unknown, it is considered sensitive.

Our main insight for recognizing authorization checks is that they must be consistently present in *all calling contexts* in which a sensitive database operation is executed. The more contexts, the better for our analysis. If we consider the context tree for each sensitive callsite separately (*e.g.*, locations 1 and 2 in Fig. 5), the callsite may have only one context (*e.g.*, location 2 in Fig. 5), making it hard to find inconsistencies. Therefore, for each sensitive operation, we combine all contexts into one list without distinguishing callsites (*e.g.*, the tree of `mysql_query` in Fig. 5).

Collecting critical variables. For each calling context associated with a sensitive database operation, SAFERPHP computes the critical variables determining reachability of this operation (see Section 4.6).

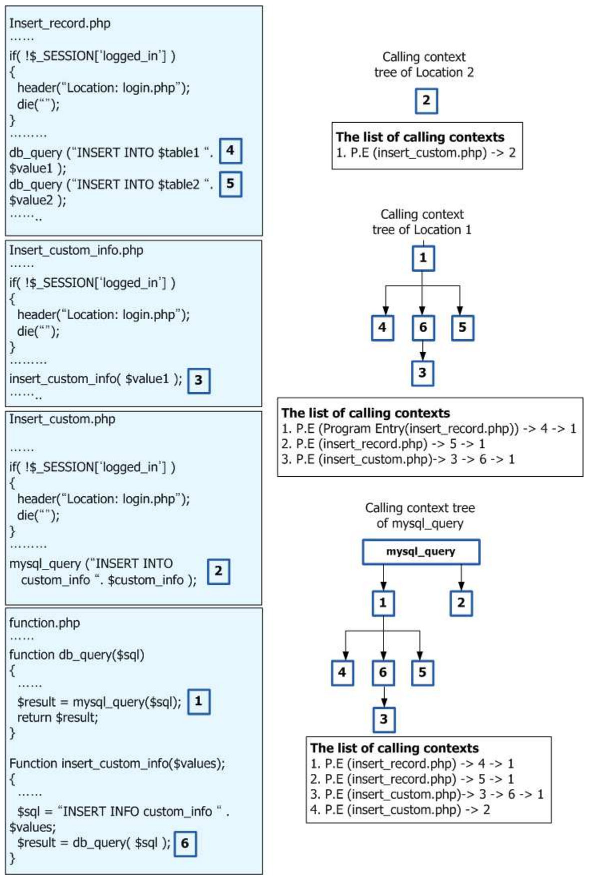
Algorithm 3 Collecting statements which may perform authorization checks

```
C : context, the set of calls whose final destination is a sensitive operation
R(C) : RESULT = set of statements which may perform security checks in context C
CD(s) : set of statements on which statement s is control-dependent
BELONG(s) : set of all statements in the function containing s statement
CALL(s) : set of callees in statement s
ALLSTATS(p) : set of all statements in function p
EXITSTATS: pre-defined set of statements die, exit, header
DOM(a,b) : true if a dominates b; false otherwise
BRANCH(s) : true if statement s is a branch

function collectTargetBlocksInContext( C )
{
   for $s \in C$ do
      for $b \in CD(s)\ \ and\ \ BRANCH(b)$ do
         R(C) ← R(C) $\cup \{b\}$
      for $stat \in BELONG(s)$ do
         if $DOM(stat, s)\ \ and\ \ CALL(stat) \neq \phi$ then
            for $callee \in CALL(stat)$ do
               visited ← ø
               collectBlocksInCallee( C, callee, visited )
}

function collectBlocksInCallee( C, p, visited )
{
   if $p \in visited$ then return
   visited ← visited $\cup \{p\}$
   for $stat \in ALLSTATS(p)$ do
      if $DOM(stat, p.exit)\ \ or\ \ DOM(stat, EXITSTATS)$ then
         if $BRANCH(stat)$ then R(C) ← R(C) $\cup \{stat\}$
         elseif $CALL(stat) \neq \phi$ then
            for $callee \in CALL(stat)$ do
               collectBlocksInCallee( callee, visited )
}
```

First, SAFERPHP collects branch statements on which each operation is control-dependent, as shown in Algorithm 3. Authorization checks may be done via function calls, thus SAFERPHP also considers every function whose callsite dominates each sensitive operation in a calling context. SAFERPHP also collects branch statements that dominate page termination calls `exit`, `die`, and `header`.

Next, SAFERPHP collects the names of variables which explicitly appear in the set of branch statements computed above. Because they affect reachability of sensitive operations, they are good candidates for the set of security-relevant variables. If b is a branch involving variable v, SAFERPHP collects all variables v_o such that $v = OP(c, v_o, b)$.

Finding inconsistencies. SAFERPHP has computed all contexts in which a sensitive database operation may be executed and the set of critical variables for each context. This set is guaranteed to include any variable which is checked prior to the sensitive operation if the check affects whether the operation is executed or not. If some variable is critical for a large fraction of contexts, but not all of them (the threshold is a parameter of the system), SAFERPHP signals a potential missing check for all calling contexts in which the variable is *not* critical. Furthemore, all calling contexts without any critical variables are flagged as vulnerabilities because in such contexts, the sensitive operation can be reached without any checks whatsoever.



```
Insert_record.php
......
if( !$_SESSION['logged_in'] )
{
  header("Location: login.php");
  die("");
}
.........
db_query ("INSERT INTO $table1 ".   4
$value1 );
db_query ("INSERT INTO $table2 ".   5
$value2 );
........
```

```
Insert_custom_info.php
......
if( !$_SESSION['logged_in'] )
{
  header("Location: login.php");
  die("");
}
.........
insert_custom_info( $value1 );   3
........
```

```
Insert_custom.php

......
if( !$_SESSION['logged_in'] )
{
  header("Location: login.php");
  die("");
}
.........
mysql_query ("INSERT INTO
   custom_info ". $custom_info );   2
```

```
function.php
......
function db_query($sql)
{
  ......
  $result = mysql_query($sql);   1
  return $result;
}

Function insert_custom_info($values);
{
  ......
  $sql = "INSERT INFO custom_info " .
$values;
  $result = db_query( $sql );   6
}
```

Calling context tree of Location 2
2
Calling context tree of Location 1
1
4
6
5
3
Calling context tree of mysql_query
mysql_query
1
2
4
6
5
3

The list of calling contexts
1. P.E (insert_custom.php) -> 2

The list of calling contexts
1. P.E (Program Entry(insert_record.php)) -> 4 -> 1
2. P.E (insert_record.php) -> 5 -> 1
3. P.E (insert_custom.php)-> 3 -> 6 -> 1

The list of calling contexts
1. P.E (insert_record.php) -> 4 -> 1
2. P.E (insert_record.php) -> 5 -> 1
3. P.E (insert_custom.php)-> 3 -> 6 -> 1
4. P.E (insert_custom.php) -> 2

Figure 5: Example of collecting contexts of sensitive database operations.

5.3 Finding other vulnerabilities

SQL injection. An ancillary benefit of taint analysis is detection of SQL injection sites. This is not our focus, but we show it to illustrate uses of the SAFERPHP framework. For every callsite, the SQL injection module checks if (1) the called function is a sensitive sink (*i.e.*, a function performing an SQL query to the back-end database), and (2) any argument is tainted. If so, SAFERPHP reports that this callsite has a risk of SQL injection and, for tracking purposes, provides the taint relation indicating the problem.

Vulnerable library functions. Many built-in PHP functions, especially in the older versions of PHP, are vulnerable to denial-of-service (DoS) attacks due to crashes or extraordinary resource consumption when called with certain inputs. These functions typically have native implementations and thus cannot be analyzed by SAFERPHP. Instead, we collected information from the National Vulnerability Database about DoS vulnerabilities reported for PHP since 2007 and compiled a list of 30 vulnerable built-in functions.

SAFERPHP checks every invocation of a function whose name is on this list and uses the taint analysis module to determine which actual arguments, if any, are tainted at this callsite. If the positions of the tainted arguments match the pre-defined vulnerable positions in the function's argument list, SAFERPHP reports that the call is vulnerable to DoS, along with the highest version of PHP affected.

Detecting vulnerable calls to built-in functions is similar to detecting SQL injection sites, but with different taint sinks and an additional check for positions of tainted arguments. For instance, the implementation of chunk_split in PHP 5.2.3 crashes if the first and third arguments are longer than 65,534 [11]. For every call to "chunk_split", SAFERPHP checks whether the first and third arguments are tainted and, if so, reports a vulnerability for PHP 5.2.3.

Misuses of uninitialized variables. Because the same variable may appear in multiple files, it may be accessed before it has been initialized. If the "register_global" option is turned on, the attacker may use a URL variable when invoking the file to provide the initial value of his choice. This is a known security issue and "register_global" is off by default since PHP version 4.2.0. Nevertheless, it may be turned on via `htaccess` file or by the administrator.

For every merged PHP file, SAFERPHP records the assignment and use positions of every variable. If (1) some use precedes the first assignment and (2) this use is in the main function, SAFERPHP flags the variable as vulnerable to injection of values via URL.

6. Experimental evaluation

We evaluated SAFERPHP by applying it to a diverse set of open-source PHP applications.

Missing authorization checks. As discussed in Section 5.2, SAFERPHP first computes the list of critical variables which affect reachability of sensitive database operations, then reports all *inconsistent* variables, *i.e.*, those checked in more than the θ fraction, but not all contexts in which the sensitive operation is executed (5th column in Table 2). The list of critical variables is further reduced by labeling (based on variable names and names of functions performing the checking) to eliminate variables unrelated to security (6th column in Table 2). During the variable labeling step, we label $_SESSION, $_COOKIE, and any variable whose name or owner function name is related to authorization as potentially relevant to security enforcement.

SAFERPHP flags all files containing calling contexts with missing checks (2nd column in Table 2). SAFERPHP also reports all calling contexts without any critical variables because they contain no checks prior to executing the sensitive operation. Overall, SAFERPHP discovered 47 previously unreported and 14 known missing-check vulnerabilities (the latter in phpCommunityCalendar).

There are several sources of false positives. First, a critical variable may be checked consistently prior to all sensitive database operations, but have no security purpose (*e.g.*, the application may access $_SESSION or $_COOKIE to re-

PHP application	No. of files with vulnerabilities	No. of false positives	No. of critical variables . . .		
			. . . appearing in $> \theta$ fraction of contexts	. . . appearing inconsistently ($> \theta$, but not all contexts)	. . . appearing inconsistently (after variable labeling)
miniblog 1.0 (LOC: 800)	0	0	6	1	0
minibloggie 1.1 (LOC: 2287)	1	0	0	0	0
DNScript (LOC: 3150)	3	0	1	1	1
seo-board 1.1.0 (LOC: 3909)	1	2	11	11	1
minibb 2.4 (LOC: 6507)	0	0	7	6	0
mybloggie 2.1.6 (LOC: 8874)	1	0	12	9	2
phpCommunityCalendar 4.0.3 (LOC: 14131)	14	1	2	2	2
php-agenda 2.2.8 (LOC: 14352)	0	1	3	3	1
minibill 1.2.5 (LOC: 19711)	16	0	0	0	0
UseBB 1.0.11 (LOC: 21073)	0	0	7	0	0
GRboard 1.8.5 (LOC: 44103)	25	9	1	1	1
NovaBoard 1.1.3 (LOC: 34260)	0	0	3	2	0

Table 2: Experimental results for missing authorization checks (θ = .35).

trieve user information). Second, applications—especially if implemented by multiple developers or evolved over time—may use different variables in different parts of the code to protect access to the same database operation. SAFERPHP will report that all such variables are used inconsistently. Third, an application may intentionally permit unchecked access to a database-modifying operation, *e.g.*, for logging or updating the number of URL clicks. Finally, SAFERPHP is occasionally unable to statically resolve the type of the SQL query performed in the database operation and may thus treat a harmless `select` as sensitive. Table 2 shows that the number of false positives is low for real-world PHP applications and can be handled manually, by human analysis of flagged variable names.

Attacker-controlled infinite loops. Results are in Table 3. SAFERPHP found one known and two new (CVE-2010-2423 and CVE-2010-2424), remotely exploitable denial-of-service (DoS) vulnerabilities, shown in Examples 2, 3, and 4. The known vulnerability occurs in PHPmailer 1.7.2 and 2.3 and is thus counted twice in the table.

Example 2 Attacker-controlled infinite loop in PHPmailer 1.7.2

```
$max_line_length = 998;
......
while(strlen($line) > $max_line_length) {
  $pos = strrpos(substr($line,0,$max_line_length)," ");
  $lines_out[] = substr($line,0,$pos);
  $line = substr($line,$pos + 1);

  if($in_headers) { $line = "\t" . $line;  }
}
```

Example 2 shows a known DoS vulnerability in PHPmailer 1.7.2 [27]. This code does not account for the possibility of strrpos returning 0. If the value of $pos is

Example 3 Another attacker-controlled infinite loop in PHPmailer 1.7.2 and PHPmailer 2.3

```
while (strlen($word) > 0) {
 $len = $length;
 if ($is_utf8) {
  $len = $this->UTF8CharBoundary($word, $len);
  } elseif (substr($word, $len - 1, 1) == "=") {
  $len--;
  } elseif (substr($word, $len - 2, 1) == "=") {
  $len -= 2;
 }
 $part = substr($word, 0, $len);
 $word = substr($word, $len);

 if (strlen($word) > 0) {
   $message .= $part . sprintf("=%s", $this->LE);
 } else {
   $buf = $part;
 }
}
```

0 and $in_header is true, the length of $line remains the same in each loop iteration. Because $line is data-dependent on user input, Example 2 is a tainted loop which the attacker may cause to execute infinitely. SAFERPHP found this vulnerability as follows. In this loop, $max_line_length and the return value of strlen($line) are exit-condition variables. At the loop head, the symbol stored in $line is SymbolArbitraryString(0,Max,{}). After the call to strrpos, $pos holds SymbolBool(false) and SymbolRangeInt(0, 998). After the third statement, $line holds SymbolBool(false) and SymbolArbitraryString(0,Max-1,{}). At the end of the loop, $line holds SymbolArbitraryString(1,Max,{"\t"}) and SymbolString("\t"). Therefore, the initial symbolic value of strlen($line) is RangeInt(0,Max) and its value after exploring one path is RangeInt(1,Max). Because these two RangeInt values overlap and the symbolic value of another exit-condition variable $max_line_length does not change,

SaferPHP concludes that this loop can be made infinite by network inputs and reports a DoS vulnerability.

Example 4 DoS vulnerability in simplehtmldom 1.1

```
function restore_noise($text) {
  while(($pos=strpos($text, '___noise___'))!==false) {
    $key = '___noise___'.$text[$pos+11].
            $text[$pos+12].$text[$pos+13];
    if (isset($this->noise[$key]))
      $text = substr($text, 0, $pos).
            $this->noise[$key].substr($text, $pos+14);
    }
    return $text;
}
```

Example 4 is a new vulnerability found by SaferPHP in simplehtmldom, a PHP library which parses HTML into DOM data structures. In function restore_noise, $text is tainted; thus, the loop is tainted. The exit-condition variable is $pos. SaferPHP found a path through the loop which does not change symbolic values of $pos and $text: any HTML attribute value containing the string "___noise___" causes an infinite loop. For instance, if a remote attacker forces the application to parse his page which contains an HTML tag: ⟨ a href="http://xyz.edu___noise___" ⟩, it will enter an infinite loop.

Example 5 Patched DoS vulnerability in PHPmailer 2.3

```
Ln1: $max_line_length = 998;
Ln2: ......
Ln3: while(strlen($line) > $max_line_length)
Ln4: {
Ln5:  $pos = strrpos(substr($line,0,$max_line_length)," ");
Ln6:
Ln7:  // Patch to fix DOS attack
Ln8:  if(!$pos) {
Ln9:   $pos = $max_line_length - 1;
Ln10:   $lines_out[] = substr($line,0,$pos);
Ln11:   $line = substr($line,$pos);
Ln12:  } else {
Ln13:   $lines_out[] = substr($line,0,$pos);
Ln14:   $line = substr($line,$pos + 1);
Ln15:  }
Ln16:
Ln17:  if($in_headers) {
Ln18:   $line = "\t" . $line;
Ln19:  }
Ln20: }
```

There are several potential sources of false positives (as Table 3 shows, their number is small in practice). First, the symbolic equivalence rules (Section 4.5) are necessarily conservative. Second, SaferPHP conservatively evaluates all static one-iteration paths, including paths that are never executed due to runtime values of branch conditions. Third, to keep analysis scalable, SaferPHP only performs intra-procedural symbolic execution. If another function or an outer loop modifies an exit-condition variable and prevents infinite looping, SaferPHP may still report a vulnerability.

Example 5 is the patched version of Example 2. In Example 2, infinite looping occurs when $pos=0. In Example 5, this causes a different branch to be taken (Lines 9-11). Statically, SaferPHP checks every one-iteration path, including the infeasible path where the second branch (Lines 13-14) is taken even though the branch condition on Line 8 is true. After Line 5, $pos holds SymbolBool(false) and SymbolRangeInt(0,998). When the second branch is symbolically evaluated with these values of $pos, the exit condition does not change, resulting in a false positive.

Uninitialized variables and vulnerable built-in functions. Table 4 shows the experimental results for finding calls to built-in functions that may cause denial of service (because the function contains a known vulnerability and a tainted argument appears in a certain position) and misuses of uninitialized variables.

A small fraction of the reported vulnerable calls to built-in functions are false positives. To cause denial of service via such a call, the attacker typically needs complete control over the tainted argument, *i.e.*, he must be able to assign an arbitrary value to it. SaferPHP does not analyze the "degree" of taintedness and conservatively considers even partially tainted values as completely controlled by the attacker. If the attacker can only partially influence the value, he may not be able to exploit the vulnerability.

For misuses of uninitialized variables, we do not rank their severity. Some misuses are critical: for example, if an uninitialized variable is used in a call to $echo$, the attacker can inject arbitrary JavaScript code into this variable. Others are less important: for example, if the uninitialized variable is the source of an assignment whose target is rarely used, it may not influence the application's behavior at all. SaferPHP simply reports all misuses of uninitialized variables, leaving it to the programmer to analyze their severity.

7. Conclusion

We characterize two new classes of semantic vulnerabilities in Web applications and present SaferPHP, a static analysis framework for discovering them in PHP source code. SaferPHP employs several new algorithms, including (i) a new type of symbolic execution to find denial-of-service vulnerabilities due to attacker-controlled infinite loops, and (ii) a new type of whole-program, inter-procedural analysis to infer the application's authorization policy and find missing checks prior to sensitive database operations. Ancillary benefits of SaferPHP include the ability to find vulnerable calls to built-in PHP functions, misuses of uninitialized variables, and SQL injection vulnerabilities. When evaluated on several open-source PHP applications, SaferPHP discovered multiple previously unreported, remotely exploitable security vulnerabilities.

Funding acknowledgments. The research described in this paper was partially supported by the NSF grants CNS-0746888 and CCF-1018271, Google research award, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

PHP application	Total number of loops	Found tainted loops	False positives	Confirmed, remotely exploitable vulnerabilities
PHPmailer 1.7.2 (LOC: 3803)	30	10	0	2
PHPmailer 2.3 (LOC: 5186)	46	16	2	1
simplehtmldom 1.11 (LOC: 1306)	48	7	4	1
phpwebthings 1.5.6 (LOC: 44017)	563	325	2	0
minibloggie 1.1 (LOC: 2287)	23	6	0	0

Table 3: Experimental results for attacker-controlled infinite loops.

PHP application	Misuses of uninitialized variables	Vulnerable built-in functions	
	Detected misuses	Detected calls	Remotely exploitable
PHPmailer 1.7.2	0	2	1
PHPmailer 2.3	0	2	1
simplehtmldom 1.11	0	0	0
phpwebthings 1.5.6	134	6	2
minibloggie 1.1	2	0	0

Table 4: Experimental results for vulnerable built-in functions and misuses of uninitialized variables.

References

[1] C. Anley. Advanced SQL injection in SQL server applications. `http://www.ngssoftware.com/papers/advanced_sql_injection.pdf`, 2002.

[2] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of Web-based applications. In *CCS*, 2007.

[3] A. Barth, C. Jackson, and J. Mitchell. Robust defenses for cross-site request forgery. In *CCS*, 2008.

[4] M. Bond, V. Srivastava, K. McKinley, and V. Shmatikov. Efficient, context-sensitive detection of real-world semantic attacks. In *PLAS*, 2010.

[5] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.

[6] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE*, 2009.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS*, 2006.

[8] R. Chang, G. Jiang, F. Ivančić, S. Sankaranarayanan, and V. Shmatikov. Inputs of coma: Static detection of denial-of-serice vulnerabilities. In *CSF*, 2009.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. *SIGPLAN Not.*, 41(6):415–426, 2006.

[10] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security*, 2003.

[11] CVE-2007-2872. `http://www.securityfocus.com/archive/1/archive/1/470244/100/0/threaded`.

[12] CVE-2009-4418. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4418`.

[13] A. Diwan, K. McKinley, and J. Moss. Using types to analyze and optimize object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 23(1):30–72, 2001.

[14] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux Security Modules framework. In *CCS*, 2002.

[15] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in Web applications. In *USENIX Security*, 2010.

[16] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, 1997.

[17] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Provig non-termination. In *POPL*, 2008.

[18] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing Web application code by static analysis and runtime protection. In *WWW*, 2004.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *S&P*, 2006.

[20] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS*, 2006.

[21] M. Kenney. Ping of death. `http://insecure.org/sploits/ping-o-death.html`, 1997.

[22] A. Klein. Cross site scripting explained. `http://crypto.stanford.edu/cs155/papers/CSS.pdf`, 2002.

[23] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *OOPSLA*, 2002.

[24] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[25] S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[26] PHC. `http://phpcompiler.org`, 2009.

[27] CVE-2005-1807. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1807`, 2005.

[28] M. Pistoia, R. Flynn, L. Koved, and V. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP*, 2005.

[29] Server Side Include (SSI) injection. `http://capec.mitre.org/data/definitions/101.html`, 2007.

[30] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: automatically inferring security specifications and detecting violations. In *USENIX Security*, 2008.

[31] G. Wasserman and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[32] WhiteHat Security. WhiteHat website security statistics report. `http://www.whitehatsec.com/home/resource/stats.html`, 2009.

[33] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security*, 2006.