Dynamic Test Input Generation for Web Applications[*]

Gary Wassermann[†]
University of California, Davis
wassermg@cs.ucdavis.edu

Dachuan Yu
DoCoMo USA Labs
yu@
docomolabs-usa.com

Ajay Chander
DoCoMo USA Labs
chander@
docomolabs-usa.com

Dinakar Dhurjati
DoCoMo USA Labs
dhurjati@
docomolabs-usa.com

Hiroshi Inamura
DoCoMo USA Labs
inamura@
docomolabs-usa.com

Zhendong Su
University of California, Davis
su@cs.ucdavis.edu

ABSTRACT

Web applications routinely handle sensitive data, and many people rely on them to support various daily activities, so errors can have severe and broad-reaching consequences. Unlike most desktop applications, many web applications are written in scripting languages, such as PHP. The dynamic features commonly supported by these languages significantly inhibit static analysis and existing static analysis of these languages can fail to produce meaningful results on real-world web applications.

Automated test input generation using the concolic testing framework has proven useful for finding bugs and improving test coverage on C and Java programs, which generally emphasize numeric values and pointer-based data structures. However, scripting languages, such as PHP, promote a style of programming for developing web applications that emphasizes string values, objects, and arrays.

In this paper, we propose an automated input test generation algorithm that uses runtime values to analyze dynamic code, models the semantics of string operations, and handles operations whose argument and return values may not share a common type. As in the standard concolic testing framework, our algorithm gathers constraints during symbolic execution. Our algorithm resolves constraints over multiple types by considering each variable instance individually, so that it only needs to invert each operation. By recording constraints selectively, our implementation successfully finds bugs in real-world web applications which state-of-the-art static analysis tools fail to analyze.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification, Reliability

Keywords

Automatic test generation, concolic testing, directed random testing, web applications

1. INTRODUCTION

Web applications continue to offer more features, handle more sensitive data, and generate content dynamically based on more sources as users increasingly rely on them for daily activities. The increased role of web applications in important domains, coupled with their interactions not only with other web applications but also with users' local systems, exacerbates the effects of bugs and raises the need for correctness.

Testing is a widely used approach for identifying bugs and for providing concrete inputs and traces that developers use for fixing bugs. However, manual testing requires extensive human effort, which comes at significant cost. Additionally, QA testing usually attempts to ensure that the software can do everything it ought to do, but it does not check whether the software can do things it ought not to do; such functionality usually constitutes security holes.

Our goal in this work is to help automate the process of web application testing. In particular, we seek to generate test cases automatically that will achieve a designated code-coverage metric: branch coverage or (bounded) path coverage. Previous work on concolic testing has helped to automate test input generation for desktop applications written in C or Java [26, 27], but web applications written in scripting languages such as PHP (ranked fourth on the TIOBE

[*] This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[†] Part of this work was performed while author was at DoCoMo USA Labs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-60558-050-0/08/07 ...$5.00.

programming community index [2]) pose different challenges that we must address.

First, PHP is a scripting language and not a compiled language. Such languages, especially in the context of web applications, encourage a style of programming that is more string- and array-centric as opposed to languages like Java where numeric values and data structures play a more central role. In the limit, scripting languages allow for arbitrary meta-programming, although most PHP programs only make moderate use of dynamic features. Additionally, PHP web applications receive all user input in the form of strings, and many string manipulation and transformation functions may be applied to these values.

Second, in order for automatic test input generation to be useful, we need test oracles that will identify when common classes of errors have occurred. Several common classes of errors in C programs are memory errors; Java has eliminated most memory errors, but Java programs may still have null-pointer dereference errors. On the other hand, PHP programs are entirely free of memory corruption errors (barring bugs in the interpreter). Hence other kinds of test oracles are needed.

Finally, much of the previous work on concolic testing is designed only for unit testing. Because many value-based and information flow-based errors (as opposed to memory-based errors, for example) span multiple functions, we need to make the concolic testing approach scale beyond single functions.

This paper presents the first application of concolic testing to web applications. We address the first challenge by modeling string operations using finite state transducers and employing a constraint resolution algorithm that resolves constraints on string values. We address the second challenge by developing a novel algorithm to check string values against existing policies to prevent SQL injection attacks [28]. We address the third challenge by using values collected at runtime to construct a backward slice from where queries are constructed and recording constraints only from that slice.

In this paper, we extend the concolic testing approach to PHP web applications. To do so, we generate constraints on string values by modeling string operations using finite state transducers (FSTs) [19]. In PHP, not only do many library functions take arguments of one type and return values of another, but the runtime system itself readily performs many different dynamic type casts. Consequently, subexpressions of the constraints we generate may be over other types including numeric types and arrays. To solve the constraints, we leverage the property of FSTs that they can be inverted and borrow from existing work on language equations [15]. The concolic testing framework helps to resolve constraints by supplying values from the program's execution in place of intractable subexpressions. In order to model precisely the semantics of PHP's many library functions and to support runtime type casts in generated constraints, we approximate expressions by considering only one variable occurrence per expression at a time. In our evaluation, none of the expressions had more than one variable occurrence, so this approximation did not introduce any imprecision. We evaluated our implementation on real-world PHP programs with known SQL injection vulnerabilities that existing static analysis tools fail to find. Our implementation took between three and thirteen minutes on these subjects and successfully found the vulnerabilities.

user.php

```
isset ($_GET['userid']) ?
  $userid = $_GET['userid'] : $userid = '';
if ($USER['groupid'] != 1)
{
  // permission denied
  unp_msg($gp_permserror);
  exit;
}
if ($userid == '')
{
  unp_msg($gp_invalidrequest);
  exit;
}
$userid = "00".$userid;
if (!eregi('00[0-9]+', $userid))
{
  unp_msg(
    'You entered an invalid user ID.');
  exit;
}
$getuser = $DB->query("SELECT * FROM"
    ."`unp user` WHERE userid='$userid'");
if (!$DB->is_single_row($getuser))
{
  unp_msg(
    'You entered an invalid user ID.');
  exit;
}
...
```

Figure 1: Example PHP code.

2. OVERVIEW

This section provides an overview of our approach.

2.1 Example Code

Figure 1 shows some sample PHP source code that we will use to present our approach. This code takes a user ID, and attempts to authenticate the user to perform other actions. If the user's ID does not appear in the database, the program exits with an error message. This particular code fragment does not use dynamic features, but it still serves to illustrate some of the main points of our algorithm. In the presence of dynamic features, we simply record the concrete values of interpreted strings and use those values in our constraint generation and resolution.

2.2 Constraint Generation

As in the case of standard concolic testing, we instrument the program in order to execute it both concretely, using the standard runtime system, and symbolically. The testing framework's top-level loop executes the program, records symbolic constraints, and uses the constraints to generate new inputs that will drive the program along a different path on the next iteration. Symbolic execution takes place at the level of a RAM machine, which means that we maintain maps from names to symbolic locations, and a map from symbolic locations to symbolic values. Therefore the analysis does not require any offline alias analysis. The testing framework records a symbolic constraint for each conditional expression that appears in the program's execution.

On the first iteration, the testing framework executes the program without providing any input parameters. When it

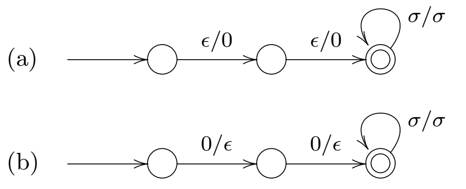

Figure 2: An FST representation of concatenating "00" to another string, and the FST's inverse; $\sigma \in \Sigma$.

encounters the `isset` conditional on line 10, it records the constraint:

$$GET[userid] \in \emptyset$$

and the program reaches line 21 and exits. Each of the constraints it gathers is expressed as a language inclusion constraint. For the next run, the testing framework inverts this constraint:

$$GET[userid] \notin \emptyset \quad \Leftrightarrow \quad GET[userid] \in \Sigma^*$$

finds ϵ, the empty string, as the shortest value in Σ^*, and reruns the program with the `_GET` parameters "userid" set to "". This illustrates a useful feature of our approach: we do not need to specify interfaces for the PHP programs we test, nor do we need a static analysis to infer them. When the program expects a parameter that our testing framework does not supply, that parameter will show up in a constraint that, when inverted, will cause the parameter to be included in the next run. This is not only the case when explicit conditionals check whether variables are set, but also when any uninitialized variables are used.

On the second iteration, the framework gathers the constraints:

$$[GET[userid] \in \Sigma^*,\ GET[userid] \in \{\epsilon\}]$$

again reaches line 21 and exits. For this example, we assume that the condition on line 12 holds. The testing framework inverts the last constraint to perform a depth-first search of the program's computation tree:

$$\begin{gathered}[GET[userid] \in \Sigma^*,\ GET[userid] \notin \{\epsilon\}] \\ \Leftrightarrow \quad GET[userid] \in \Sigma^+\end{gathered}$$

Again the testing framework selects some shortest value in Σ^+, in this case '`a`.'

On the third iteration, the framework gathers the constraints:

$$\begin{gathered}[GET[userid] \in \Sigma^*,\ GET[userid] \notin \{\epsilon\}, \\ 00.GET[userid] \notin \mathcal{L}(A_{\texttt{00[0-9]+}})]\end{gathered}$$

Inverting constraints such as the last one here requires techniques beyond those that have been proposed in the literature because this constraint includes a string operation, *viz.* concatenation.

2.3 Constraint Resolution

The problem of satisfiability of word equations with regular constraints is PSPACE-complete [23]. However, our constraint language is more expressive than this because we include nondeterministic rational relations, expressed as FSTs, and many classes of language constraints are undecidable [15]. Consequently, we cannot solve precisely every constraint in the language of constraints that may be generated. However, a benefit of the concolic testing framework is that the constraint resolution algorithm can be incomplete or even wrong, and no false positives will be reported. We design our algorithm for the common case in which input variables appear only on the left-hand side of each language inclusion constraint.

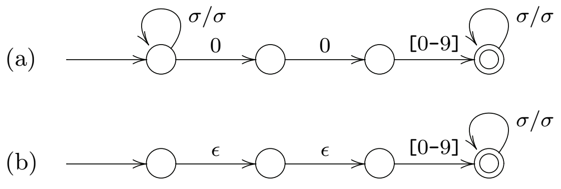

Figure 3: An FSA representation of the `ereg` "`00[0-9]+`" and its image over the FST in Figure 2b; $\sigma \in \Sigma$.

As Section 1 states, our algorithm uses finite state transducers (FSTs) to invert string operations. Figure 2a shows an FST the represents the curried function "`00.`", *i.e.*, the function that prepends the string "`00`" to its argument. The first two transitions each read nothing and output "`0`" and the third transition outputs whatever it reads. FSTs can be inverted by swapping the input symbol with the output symbol on each transition. Figure 2b shows the FST inverted.

Figure 3a shows an FSA representation of the language of strings that match the regular expression on line 24 of Figure 1. Because the regular expression does not have anchors ('`^`' for "beginning of the string" and '`$`' for "end of the string"), the pattern must only appear somewhere in the string, as the FSA shows. Because the FST in Figure 2b represents the inverse of prepending "`00`," this FST can be applied to the FSA in Figure 3a to produce the FSA in Figure 3b. The language of this FSA represents the language of values for $GET[userid]$ for which the conditional expression on line 24 will evaluate to true. As before, the language of this FSA must be intersected with the languages of the other FSAs for the same variable in order to find the language of values that will cause the program to take a new path in its computation tree. A new value, such as "`0`" can then be selected for the `userid` `GET` parameter.

2.4 Test Oracles

In order to be useful, automatic test input generation requires a *test oracle* that will give feed back on each execution of the program. Typically this feedback takes the form of *pass* or *fail.* In the case of testing C programs, example test oracles include `assert` statements, and tools like Valgrind [20], which monitors the memory and checks for memory errors such as buffer overflows and double-frees. Scripting languages, however, are not vulnerable to memory corruption errors, and although programmers use `exit` statements, in practice, very few use `assert`'s.

The same need for test oracles arises with fuzz testing and testing based on a preset list of inputs, and currently at least two kinds of oracles are used. First, security testers often see whether the input causes the web browser to pop up an alert window. If it does, this indicates a cross-site scripting (XSS) vulnerability. Second, testers check to see whether corresponding pages of sites written to be configured for multiple natural languages have the same structure. If they do not, this indicates that some data is missing in one of the languages.

These test oracles are available for our setting as well, but because our testing framework has full view of all the data values, we can use a more sophisticated oracle. Grasp [9] is a modified version of the PHP interpreter that performs character-level tainting [21], and allows security policies to be defined on strings based on tainting. A typical example of such a policy defines SQL injection attacks as SQL queries in which characters in tokens other than literals are tainted, or more generally, only syntactically confined substrings are tainted [28]. Given the SQL grammar (CFG) $G = (V, \Sigma, R, S)$ and a query string $\sigma = \sigma_1\sigma_2\sigma_3 \in \Sigma^*$, σ_2 is *syntactically confined* in σ iff there exists a sentential form $\sigma_1 X \sigma_3$ such that $X \in V$ and $S \Rightarrow_G^* \sigma_1 X \sigma_3 \Rightarrow_G^* \sigma_1\sigma_2\sigma_3$ [29]. We can also use other taint-based policies, such as the policy that tainted strings in the web application's output document must not invoke the clients JavaScript interpreter. This is a less heuristic approach to finding XSS vulnerabilities.

An advantage to using taint-based policies such as the ones described above is that we can attempt to generate inputs that will result in failing runs and so discover bugs. In the case of SQL injection vulnerabilities, prior to each call to the query function in the database API, we apply an implicit *SQL conditional* to the string value of the query. That conditional does not appear in the program or in the execution of the program; it is simply recorded as a constraint in our symbolic execution. The constraint specifies that substrings in the query from user input are syntactically confined. In order to invert this constraint, we construct a transducer that inverts the operations that constructed the query string, just as before. We then construct the image of the SQL CFG over that transducer. The image of a context-free language represented by a CFG over an FST can be constructed using an adaptation of the CFL-reachability algorithm [24] to construct the intersection of a CFG and an FSA [12].

The structure of the resulting CFG G' corresponds to the structure of the SQL CFG such that for a PHP variable v whose value is used to construct the query string, we extract a sub-grammar G_v from G' such that $v \in \mathcal{L}(G_v)$. The SQL predicate checks symbolically whether all possible values for v are safe based on the structure of G_v. In the case of our running example, this predicate does not hold. Rather than inverting it by taking its complement, we extract from G_v G'_v, the grammar for values for v where $\mathcal{L}(G'_v) \subseteq \mathcal{L}(G_v)$ and every string in $\mathcal{L}(G'_v)$ represents an attack input. G'_v can be constructed because it is based on the structure of G_v.

To resolve the constraints on v, we take the intersection of $\mathcal{L}(G'_v)$ with the intersection of the other regular languages that bound v's range. The result is a CFG, and finding a word in the language of a CFG can be done in linear time. Such a word will then be supplied as input for the next test run, and if it indeed violates the security policy, the runtime system will catch it. Because the intersection of two CFGs cannot be constructed in general, we only handle one SQL constraint on each variable at a time. In the case of our running example, the result of resolving the SQL predicate is too involved to show in a meaningful picture, but the algorithm will produce a string like "`0' OR 'a'='a`," which will result in an attack.

$$
\begin{array}{lll}
e & \rightarrow & e\ \mathit{bin\_op}\ e \quad | \quad \mathrm{c} \quad | \quad (\mathit{cast})\ e \\
 & | & f(e,\ldots) \quad | \quad e[e] \quad | \quad v \\
\mathit{bin\_op} & \rightarrow & \mathit{str\_op} \mid \mathit{num\_op} \mid \mathit{bool\_op} \\
f & \rightarrow & \mathit{str\_f} \mid \mathit{num\_f} \mid \mathit{bool\_f} \mid \mathit{arr\_f} \\
\mathit{str\_op} & \rightarrow & . \quad \text{(concatenation)} \\
\mathit{str\_f} & \rightarrow & \text{trim} \mid \text{add\_slashes} \mid \text{implode} \mid \ldots \\
\mathit{num\_op} & \rightarrow & + \mid - \mid \times \mid \div \mid \% \\
\mathit{num\_f} & \rightarrow & \text{count} \mid \text{strlen} \mid \ldots \\
\mathit{bool\_op} & \rightarrow & = \mid \neq \mid > \mid \geq \mid < \mid \leq \\
\mathit{bool\_f} & \rightarrow & \text{isset} \mid \text{is\_int} \mid \text{ereg} \mid \ldots \\
\mathit{arr\_f} & \rightarrow & \text{explode} \mid \text{split} \mid \text{array\_combine} \mid \ldots \\
\mathit{cast} & \rightarrow & \text{int} \mid \text{float} \mid \text{string} \mid \text{bool} \mid \text{array}
\end{array}
$$

Figure 4: Expression language.

2.5 Selective Constraint Generation

In real-world programs, much of the program's execution has little to do with the property of interest. For example, a web application page that constructs a query may also instantiate a timer class to limit the execution time of the page, construct from configuration files HTML fragments that have text in the users language, and other such operations. Gathering constraints on unrelated parts of the code adds unnecessary overhead both to the constraint generation and to the constraint resolution. Previously, however, automated input test generation efforts did gather constraints for the entire execution because: (1) the points of possible failure may not be known statically; and (2) it may be difficult to compute a backward static slice from failure points that are known.

We propose an iterative approach to narrow the focus of our constraint generation to constraints that are relevant to possible failures. First, we identify points of possible failure. In the case of SQL injection, these are the program points where the API function is called to send a query to the database. We then add all functions in which these points occur to a set of functions to be analyzed. We then execute the program (*i.e.*, load the page) and gather constraints from only the functions in this set. If the initial execution does not encounter a query function and hence gathers no constraints, we instrument the top-level file and in successive iterations instrument included files until a query function is encountered. We resolve control dependencies by recording a stack trace at the beginning of the function call. Those functions that invoked the analyzed functions then get added to the set of functions to analyze. We resolve data dependencies by examining symbolic values, for example, the symbolic value of the constructed query, and where branches terminate at function calls, we add those functions to the set as well. We then re-execute the program gathering constraints from all of the functions in the set, and repeat this process until no more dependencies exist. This process approximately computes a backward slice, but we use runtime data to compute it even in the presence of dynamically constructed function call names and variable names. By gathering constraint selectively, we reduce by several orders of magnitude the size of the constraint set to gather and resolve. Reducing the constraint set size facilitates scaling to larger programs, as our evaluation shows.

```
SingleOcc($t$)
switch $t$
   case $v$ :
         return $\{t\}$
   case $c$ :
         return $\{\}$
   case $(cast)e$ :
         $r \leftarrow \emptyset$
         $ss \leftarrow$ SingleOcc($e$)
         for each $s$ in $ss$
         do $r \leftarrow r \cup \{(cast)s\}$
         return $r$
   case $e_1\ bin\_op\ e_2$ :
         $r \leftarrow \emptyset$
         $s_1 \leftarrow$ SingleOcc($e_1$)
         $c_2 \leftarrow$ GetConcrete($e_2$)
         for each $s$ in $s_1$
         do $r \leftarrow r \cup \{s\ bin\_op\ c_2\}$
         $s_2 \leftarrow$ SingleOcc($e_2$)
         $c_1 \leftarrow$ GetConcrete($e_1$)
         for each $s$ in $s_2$
         do $r \leftarrow r \cup \{c_1\ bin\_op\ s\}$
         return $r$
   case $f(args\ list)$ :
         $r \leftarrow \{\}$
         $cc \leftarrow$ map GetConcrete $args$
         for $i$ in $0$ to $length(args)$
         do $s_1 \leftarrow$ SingleOcc($args[i]$)
            if $\emptyset \neq s$
               then $cs \leftarrow cc$
                    $cs[i] \leftarrow s_1$
                    for each $s$ in $s_1$
                    do $r \leftarrow r \cup \{f(cs)\}$
         return $r$
   case ... :
```

Figure 5: Algorithm to construct sets of single variable-occurrence expressions.

3. ALGORITHM

Figure 4 gives the grammar for the Boolean expressions from a PHP-like language, and the grammar implicitly defines the structure of the expressions' abstract syntax trees. The grammar includes some representative functions that return values of types string, Boolean, and array, as well as values of numeric types. The constraints recorded from the execution of the subject program come from this grammar. Although the grammar does not specify the arity of each function, PHP's runtime system executes only programs in which functions have the right number of arguments. Since we analyze constraints collected from a run of the program, the PHP runtime system guarantees that each function will be passed the right number of arguments.

The PHP runtime performs runtime casts among many types automatically. For example, PHP will not throw an exception when trying to execute $(3 + \text{“a”})$; rather, because the "+" operator takes two integers as arguments, the runtime system will automatically cast "a" to an integer and evaluate the expression. Because the string "a" does not represent a numeric value, the runtime system will cast "a" to 0. In addition, many functions, such as the length function, take arguments of one type and return arguments of another.

c	$\rightarrow$	$\bullet \in \mathcal{L}$	$\mathcal{L}$ is a regular language
	$\mid$	$\bullet\ op\ i$	integer
	$\mid$	$\bullet\ op\ f$	float
	$\mid$	$\bullet = b$	Boolean
	$\mid$	$\bullet$*[index]* c	constraint on array element
	$\mid$	$\lvert \bullet \rvert\ op\ i$	constraint on array length
index	$\rightarrow$	i	integer index
	$\mid$	s	string index
	$\mid$	$\top$	unknown index
op	$\rightarrow$	$= \mid \neq \mid > \mid \geq \mid < \mid \leq$	

Figure 6: Constraint language.

Previous work on concolic testing solves constraints by selecting a theory and replacing expressions that lie outside of the chosen theory with the corresponding concrete value gathered from an execution. PHP's propensity to convert between types makes it difficult to identify a decidable theory that covers most of the constraints that a program execution will generate. Consequently, we adopt a different approximation strategy from the approach proposed in previous work on concolic testing. For each variable occurrence in a Boolean control expression encountered in a test execution, we create a copy of the expression and set all other variable occurrences in the expression to their concrete values from the execution. The SingleOcc function in Figure 5 shows how this operation is performed on a representative set of expressions. It returns a set of expressions where each expression in the set has only a single variable occurrence and each subexpression that does not depend on that variable is replaced with its concrete value. Note also that certain subexpressions, such as function names, are replaced automatically with their concrete values.

The constraint gathering phase collects a list of Boolean control expressions along with their runtime values, each one being either true or false. In order to generate a set of input values to take a new program path, we select from the list an expression e to invert and discard the expressions following e in the list. We apply SingleOcc to each expression in the remaining list. The result is a list of expressions, each with a single variable occurrence, and each with a Boolean value to which it ought to evaluate in order for the new input values to drive execution along the designated path.

The function SolveFor in Figure 11 generates a constraint on the variable in an expression by solving the expression against a constraint. We pass to it an expression from the list of Boolean control expressions along with the expected Boolean value as a constraint. Figure 6 shows the language of constraints for this algorithm. Each constraint form has a hole, designated with "$\bullet$", for the expression it constrains. The language includes constraints on integer, decimal, Boolean, string, and array values. The constraints on Boolean values are simply equality to Boolean constants. When SolveFor is called with an initial expression and Boolean value, the constraint says that the value of the expression equals the given Boolean value. The constraints on string values are regular language membership constraints. The constraints on arrays include length constraints and element constraints. The language does not, however, include

cast	true	false
(int)	1	0
(float)	1.0	0.0
(string)	“1”	“”
(array)	[true]	[false]

Figure 7: PHP’s rules for type conversion from Boolean.

```
MakeBoolConstraint($t$)
 1  switch $t$
 2      case $b$ : return $b$
 3      case $op\ n$ when $\neg(0\ op\ n)$ and $\neg(1\ op\ n)$ :
 4              raise Unsatisfiable
 5      case $op\ n$ when $0\ op\ n$ and $\neg(1\ op\ n)$ :
 6              return false
 7      case $op\ n$ when $\neg(0\ op\ n)$ and $1\ op\ n$ :
 8              return true
 9      case $op\ n$ when $0\ op\ n$ and $1\ op\ n$ :
10              raise Unconstrained
11      case $op\ n$ when $x\ op\ n \Rightarrow x \neq 0$ : return true
12      case $\in \mathcal{L}$ when $1 \notin \mathcal{L}$ and $\epsilon \notin \mathcal{L}$ :
13              raise Unsatisfiable
14      case $\in \mathcal{L}$ when $1 \in \mathcal{L}$ and $\epsilon \notin \mathcal{L}$ : return true
15      case $\in \mathcal{L}$ when $1 \notin \mathcal{L}$ and $\epsilon \in \mathcal{L}$ : return false
16      case $\in \mathcal{L}$ when $1 \in \mathcal{L}$ and $\epsilon \in \mathcal{L}$ :
17              raise Unconstrained
18      case $length\ op\ i$ when $0\ op\ i$ :
19              raise Unconstrained
20      case $length\ op\ i$ : raise Unsatisfiable
21      case $[index]t'$ : return MakeBoolConstraint($t'$)
```

Figure 8: Algorithm for converting arbitrary constraints to Boolean constraints.

standard Boolean operators, such as conjunction and disjunction, because it is designed for expressions with a single variable occurrence.

The SolveFor function is designed to retain as much precision as possible across arbitrary type conversions. It does not require that expressions have the same type as the supplied constraint. Rather, it converts the constraint to have the same type as the expression. Converting the type of the constraint requires working backward across PHP’s type conversion rules. To illustrate, Figure 7 shows the rules for converting Boolean values to integer, floating point, string, and array values. Figure 8 shows the rules for converting an arbitrary constraint. The following contrived example helps to illustrate this point:

step	expression	constraint
1	5 + ereg(“ˆ[a-z]*$”, x)	(= 5)
2a	ereg(“ˆ[a-z]*$”, x)	(= 0)
2b	ereg(“ˆ[a-z]*$”, x)	(false)
3	x	$\left(\overline{\mathcal{L}(\text{ˆ[a-z]*\$})}\right)$

The expression and the constraint in step 1 both have type integer, so the 5 can be subtracted from both. The ereg function in step 2a returns a Boolean value, but the constraint is over integers. According to the type conversion rules in Figure 7, false converts to 0, which satisfies the constraint, and true converts to 1, which does not. The constraint then gets converted to false in step 2b. The function ereg returns false when the value of its second argument is not in the language represented by its first argument, so step 3 finishes with the variable x as its expression and a constraint specifying a language in which the value of x lies.

```
Image($A, F$)
    // Image of FSA $A$ over FST $F$
    $(Q_1, \Sigma, s_1, f_1, \delta_1, L_1) \leftarrow A$
    $(Q_2, \Sigma, s_2, f_2, \delta_2, L_2) \leftarrow F$
    $\delta \leftarrow \emptyset; Q \leftarrow \emptyset; L \leftarrow \emptyset$
    $\delta'_1 \leftarrow \delta_1; \delta'_2 \leftarrow \delta_2$
    for each $q \in Q_1$
    do $\delta'_1 \leftarrow \delta'_1 \cup \{(q, \epsilon, q)\}$
    for each $q \in Q_2$
    do $\delta'_2 \leftarrow \delta'_2 \cup \{(q, \epsilon, \epsilon, q)\}$
    for each $(q_{s1}, i, q_{t1})$ in $\delta_1$
    do for each $(q_{s2}, i, o, q_{t2})$ in $\delta_2$
         do $\delta \leftarrow \delta \cup \{((q_{s1}, q_{s2}), o, (q_{t1}, q_{t2}))\}$
    for each $q_1$ in $Q_1$
    do for each $q_2$ in $Q_2$
         do $Q \leftarrow Q \cup \{(q_1, q_2)\}$
             if $q_1 \in$ domain $(L_1)$
               then $L \leftarrow L \cup \{(q_1, q_2) \rightarrow L_1(q_1)\}$
    $(Q, \Sigma, (s_1, s_2), (f_1, f_2), \delta, L)$
```

Figure 9: FSA image construction.

```
InvImage($a$ : FSA, $f$ : FST)
    $a_2 \leftarrow$ Complement($a$)
    $f_2 \leftarrow$ Inv($f$)
    $a_3 \leftarrow$ Image($a_2, f_2$)
    $a_4 \leftarrow$ Complement($a_3$)
    return $a_4$
```

Figure 10: Algorithm to find the pre-image of an FSA over an FST.

As Figure 8 shows, certain type conversions lose all constraint on the value of the expression (as on lines 10 and 17), and other type conversions are inconsistent with the constraint (as on lines 4 and 13). When inconsistencies occur, the algorithm fails to find input values for a certain path. Converting constraints to Boolean constraints does not require approximation, but converting to other types requires approximation in some cases.

Figure 9 gives an algorithm for finding the image of an FSA over an FST. Figure 10 shows the algorithm that calls Image to find the maximal pre-image of a regular language over an FSA. The SolveFor function uses the InvImage routine for solving over PHP’s string functions, as on line 19 in Figure 11. Because FSTs may be nondeterministic, applying the inverse of an FST to an FSA directly will not yield the maximal pre-image; the complement of the pre-image of the complement of the language yields the maximal solution.

Section 2.4 describes an artificial SQL predicate that our instrumentation inserts into programs wherever the SQL query function is called. This predicate initially takes the form of “$\bullet \in \mathcal{L}(G_{SQL})$,” where G_{SQL} is the SQL grammar. Resolving this predicate determines whether SQL injection attacks are possible, and if so, generates input that will cause

```
SolveFor($e$ : expr, $t$ : constraint)
switch $e$
   case $c$ :
         raise ConstantExpression
   case $v$ :
         return $v, t$
   case $e' + c$ :
         $op\ i \leftarrow$ MakeIntConstraint($t$)
         $t' \leftarrow op\ (i - c)$
         return SolveFor($e', t'$)
   case $ereg(reg, e')$ :
         $b \leftarrow$ MakeBoolConstraint($t$)
         $fa \leftarrow$ RegToFsa($reg$)
         if not $b$
            then $fa \leftarrow$ Complement($fa$)
         $t' \leftarrow \in \mathcal{L}(fa)$
         return SolveFor($e', t'$)
   case $stripslashes(e')$ :
         $\in \mathcal{L}(fa) \leftarrow$ MakeStrConstraint($t$)
         $fa \leftarrow$ InvImage($fa, f_{stripslashes}$)
         $t' \leftarrow \in \mathcal{L}(fa)$
         return SolveFor($e', t'$)
   case $count(e')$ :
         $t_1 \leftarrow$ MakeIntConstraint($t$)
         $t_2 \leftarrow length\ t_1$
         return SolveFor($e', t'$)
```

Figure 11: Algorithm to solve for variables.

an attack. The algorithm for resolving this predicate proceeds initially as the algorithm for resolving other predicates: operations on the single occurence of a variable are successively inverted and applied to the grammar. The image of a context free grammar over a finite state transducer is again context free. We need only consider each query site individually, so other SQL predicates are excluded from the symbolic expression and the intersection of two context-free grammars will not be needed. This resolution produces a predicate that has the form "$v \in \mathcal{L}(G)$" for some grammar G. If other regular language predicates on the variable v exist, the predicates can be combined by computing the intersection of the regular languages and G.

The CFG representation G_2 of the image of a CFG G_1 over an FST or the intersection of a CFG G_1 and an FSA can be constructed such that each nonterminal in G_2 corresponds to some nonterminal in G_1 (*cf*, Figure 7 in [29]). This means that for a predicate "$v \in \mathcal{L}(G)$," the symbols in G can be related to the symbols in the SQL grammar G_{SQL}. Let $P_{SQL}(X_1) = X_2$ if X_1 is a nonterminal in G, X_2 is a nonterminal in G_{SQL}, and X_1 was constructed based on X_2. The grammar G then describes how the initial values of v, after being transformed by the program and incorporated into queries, will relate to the SQL grammar. G is constructed conservatively, so that it may describe some initial values of v as being incorporated into queries in ways that they will not.

Let G be *precise* if (1) every string in $\mathcal{L}(G)$ is a value for v for which the program will take the same path as in the logged execution provided that the other inputs remain the same, and (2) every string value, after being transformed and incorporated into a query, will be parsed under G_{SQL} as given by the correspondence between the symbols in G and the symbols in G_{SQL}. If $G = (V, \Sigma, R, S)$ is precise, then a string $s \in \mathcal{L}(G)$ will cause an attack iff there exists no X such that the following conditions hold:

- $s \in \mathcal{L}(V, \Sigma, R, X)$ and
- for all sententials forms γ such that $X \Rightarrow^* \gamma \Rightarrow^* s$, if $\gamma = \alpha X_1$ or $\gamma = X_1 \alpha$, and if $\alpha \Rightarrow^* s$, then $P_{SQL}(X_1) \Rightarrow^*_{G_{SQL}} \epsilon$.

Such a string $s \in \mathcal{L}(G)$ exists for $G = (V, \Sigma, R, S)$ if for some $X \in V$ and some $X_1 \rightarrow \alpha X \beta$, $\mathcal{L}(V, \Sigma, R, X) \not\subseteq \{\epsilon\}$ and either

- $\beta = \epsilon$, $FOLLOW(X) \not\subseteq \{\epsilon\}$, and $\alpha \Rightarrow^* X_2 \alpha'$, or
- $\alpha = \epsilon$, $PRE(X) \not\subseteq \{\epsilon\}$, and $\beta \Rightarrow^* \beta' X_2$,

where $\epsilon \in \mathcal{L}(V, \Sigma, R, X_2)$ and $\epsilon \notin \mathcal{L}(V_{SQL}, \Sigma_{SQL}, R_{SQL}, P_{SQL}(X_2))$, and where $FOLLOW(X)$ is the follow set of X and $PRE(X)$ is the follow set of X in the right-to-left direction. Standard algorithms exist for finding the follow set of a nonterminal and determining whether a nonterminal can derive ϵ. In order to generate a string that will cause an attack, we identify an X as described above and derive a string through it, including a non-empty string from X's follow/pre set and ϵ derived from X_2.

4. EVALUATION

This section discusses our implementation and the test cases we used, and then presents the results of our evaluation.

4.1 Implementation

As previously discussed, our approach has two phases: constraint generation and constraint resolution. PHP is an interpreted language, so the constraint generation phase could be implemented directly in the interpreter. However, it is not clear how the first phase could avoid generating unnecessary constraints if the interpreter has only the web application code. Consequently, we chose to implement constraint generation at the language level. We wrote a plugin to phc, an open source PHP compiler front-end [7], to perform a source-to-source transformation on the PHP code that we want to gather constraints from. The plugin consists of about 2200 lines of C++, and it wraps each statement in a function call. The wrapper functions write to a file a trace log of the program execution. Additionally, on evaluated strings in transformed code, the transformed program first passes the code to be executed through the source-to-source transformation so that the new code will also be logged.

The second phase reads in the log and symbolically executes the trace. It produces a list of Boolean control expressions where each subexpression is annotated with a concrete value from the execution. This phase is implemented using about 5200 lines of OCaml in addition to Minamide's finite automata and regular expression libraries [19].

4.2 Test Subjects

We selected three real-world PHP web applications with known SQL injection vulnerabilities to evaluate our implementation. The first, Mantis 1.0.0rc2, is an open source bug tracking system, similar to Bugzilla. It has an SQL injection vulnerability in its "lost password" page, and the top-level PHP file for this page includes transitively 27 other files for

gpc_api.php

```
46  function
47  gpc_get_string( $p_var_name,
48                  $p_default = null ) {
49
50    $args = func_get_args();
51    echo "args = ";
52    $t_result =
53      call_user_func_array( 'gpc_get',
54                            $args );
55
56    if ( is_array( $t_result ) ) {
57      error_parameters( $p_var_name );
58      trigger_error(
59          ERROR_GPC_ARRAY_UNEXPECTED,
60          ERROR );
61    }
62
63    return $t_result;
64  }
```

Figure 12: Input handling code in Mantis.

a total of 17,328 lines of PHP in the page. The second, Mambo 4.5.3, is an open source content management system. It has an SQL injection vulnerability in its "submit weblink" page, and the top-level PHP file for this page includes transitively 23 other files for a total of 13,248 lines of PHP in the page. The third, Utopia News Pro 1.3.0, is a news management system. It has an SQL injection vulnerability due to insufficient regular expression filtering in its user-management page. It includes transitively 6 other files for a total 1,529 lines of PHP.

Both of the first two web applications we tested use dynamic features for parts of the code that are relevant to query construction. First, both web applications include files dynamically by specifying the names of files to include via dynamically constructed string values. Some static analyzers require user intervention to provide static file names in order to get all of the code that the application will use, although others use constant propagation or related techniques to construct some file names automatically.

Second and more importantly, both use dynamic features in handing user input. Figure 12 shows the gpc_get_string function from Mantis' gpc_api.php file ("gpc" stands for GET-POST-COOKIE, the three primary vehicles for delivering user input to the application server). The call to func_get_args() on line 50 returns as an array the list of arguments that was passed to the user-defined function in which it is called. The call to call_user_func_array() on line 53 calls the function named by the string value of the first argument passing array in the second argument the function as an argument list. In this case, it is the function gpc_get that retrieves input values directly. Figure 13 shows the mosGetParam function, the function used for getting input values, from Mambo's mambo.php file. This function takes as arguments an array reference and the string value of the name of an index and returns the value of the appropriate array element. In some calls to this function, the array passed is itself a dynamically index element of an array (*e.g.*, GLOBALS).

We tried to analyze both web applications using two static analyzers: Pixy [14], and Wassermann and Su's tool [29], which is based on Minamide's PHP static analyzer [19]. Because of dynamic features such as those shown above, both failed to find any SQL injection vulnerabilities.

mambo.php

```
function
mosGetParam( &$arr, $name, $def=null,
             $mask=0 ) {
  if (isset( $arr[$name] )) {
    if (is_array($arr[$name]))
      foreach ($arr[$name] as
               $key=>$element)
        mosGetParam ($arr[$name], $key,
                     $def, $mask);
    else {
      if (!($mask&_MOS_NOTRIM))
        $arr[$name] =
            trim( $arr[$name] );
      if (!is_numeric( $arr[$name] )) {
        if (!($mask&_MOS_ALLOWHTML))
          $arr[$name] =
            strip_tags($arr[$name]);
        if (!($mask&_MOS_ALLOWRAW)) {
          if (is_numeric($def))
            $arr[$name] =
                intval($arr[$name]);
        }
      }
    }
    return $arr[$name];
  } else {
    return $def;
  }
}
```

Figure 13: Input handling code in Mambo.

Test Case	Translated functions	Log file size	Logging time (s)
Mantis	1	8 KB	1
	2	13 KB	1
	3	18 KB	1
	4	19 KB	1
	all	≥ 2.9 GB	≥ 600

Figure 14: Trace log file data.

4.3 Evaluation

As previously stated, the first phase of our analysis performs a source-to-source translation on PHP files so that the resulting files, when executed, will write to a file a trace log of their execution. For all execution and logging experiments, we set the maximum execution time at 5 minutes per iteration (execute, log, resolve constraints). The first set of experiments we ran shows demonstrates that logging the whole trace can be prohibitively expensive. Figure 14 shows the execution times and corresponding log sizes for Mantis when increasing numbers of functions were translated and executed. For the assertion we were checking, four functions proved to be sufficient to cover the backward slice. When all of the code was translated and executed, the page failed to load in our browser before timing out. At that point, the log file size was 2.9 GB. In this experiment, when a file was to be included, our translation dynamically translated the file, wrote it to a new file, and included the new file. This

Test Case	Inputs	Time	Max Log
	Generated	(mm:ss)	Size (KB)
Mambo	4	13:02	65
Mantis	5	03:38	19
Utopia News Pro	23	05:14	17

Figure 15: Iterations to find an injection vulnerability.

dynamic translation added to the execution time, but not to the log file size.

In our experiments, only the conditional expressions on constructed queries in our added assert statements had more than one variable occurrence. This means that for the conditional expressions in our experiments, our algorithm did not make any approximations by considering only one variable occurrence per expression instance.

Figure 15 shows for each program in our evaluation how long it took to find an input that caused an SQL injection attack in terms of the number of test inputs generated and the total time to generate them. Mambo and Mantis required relatively few test inputs before they generated an attack. This is because in each of their pages, the query constructed at the vulnerable program point plays a central role in the page. If the inputs that get included in the query are not present, the page produces an error message before it has done much else. Once the inputs are provided that cause the page to produce a query successfully, the execution also encounters the implicit conditional inserted by our source-to-source transformation that checks whether the query is an attack. The next input will then produce an attack. In contrast, our implementation produced 22 inputs to Utopia News Pro before producing one that results in an attack. This is because the page we tested performs several roles in the application and essentially has a large switch-case statement on input values to select which action it should take. The vulnerable program point in this page was not reached until several other branches of the switch-case statement had been tried. Although Mambo required the fewest inputs of our test cases to reach an injection attack, it took the most time. This was because, as indicated by the maximum trace-log size, the page performed more operations before reaching the vulnerable program point than it did for Mantis or Utopia News Pro. Consequently, there were more constraints to be resolved for each path, and for each input generated, our implementation attempted to resolve constraints for several paths that proved to be unsatisfiable.

5. LIMITATIONS

This section discusses some limitations of our approach.

Previous work on leveraging symbolic and runtime values for input test generation falls back on concrete values when the symbolic value being addressed lies outside the theory of the resolution algorithm's decision procedure. Our constraint resolution algorithm generates constraints only based on one variable instance per value. Therefore it may under-approximate the symbolic values of variables when program predicates depend on multiple variables, and it may miss paths that other resolution algorithms would find. In principle our constraint resolution algorithm could be enhanced to include multivariate constraints in some cases, but we leave that to future work.

Our approach of logging files selectively is effective only when the points of possible failure are known and relatively localized, as is the case with SQL injection, where the possible failure point is where the program sends queries to the database. If the problems of interest are potentially more ubiquitous in the program code, as with arbitrary runtime exceptions, logging selectively will be less effective. Logging the whole execution trace would address that problem, but it is prohibitively expensive. We expect that modifying the PHP interpreter to generate symbolic constraints directly may alleviate some of the expense of execution time, but that may make selective logging difficult.

At present, our implementation is not fully automated. The web page must be manually loaded (*e.g.*, by clicking "go"), the analyzer must be manually invoked, and analyzer writes the next inputs to a file, so they must be manually provided to the URL. However, in principle, nothing about our approach requires user interaction.

6. RELATED WORK

In this section, we survey closely related work.

6.1 Test Input Generation

Traditional work on testing has generated random values as inputs [6, 17, 22]. Randomly generated input values will often be redundant and will often miss certain program behaviors entirely. Test input generation that leverages runtime values, or concolic testing, has been pursued by multiple groups [3, 4, 5, 10, 26, 27]. These approaches gather both symbolic constraints and concrete values from program executions, and use the concrete values to help resolve the constraints to generate the next input. Previous work on concolic testing handles primarily constraints on numbers, pointer-based data structures, and thread interleavings. This is appropriate for the style of programming that languages like C and Java encourage, but scripting languages, especially when used in the context of web applications, encourage a style in which strings and associative arrays play a more central role.

Perhaps the work most closely related to ours is by Emmi et al., in which they augmented concolic testing to analyze database-backed Java programs. They added support for string equality and inclusion in regular languages specified by SQL `LIKE` predicates [8]. Our work is distinguished from theirs in at least the following aspects. They support a form of multi-lingual programming in which Java programs generate SQL queries, whereas we support a setting in which more general meta-programming is possible. They do not support any string operations, although they mention that string constraints with concatenation can resolved in PSPACE; where as we support concatenation as well as many other string operations that PHP provides, although this requires us to make some approximations in our constraint resolution algorithm. They check for the same properties as standard concolic checking, whereas we check for security problems common among web applications.

6.2 Web Application Testing

Some previous work on web application testing has focussed on static webpages and the loosely structured control flow between them (defined by links), and other work has focussed on the server-side code, often carrying over techniques from traditional testing. Early work on web application focussed primarily on static pages and the coverage metric was page-coverage. Ricca and Tonella propose a technique for using UML models of web applications to analyze static web pages via testing [25]. Kung et al. model web applications as a graph and develop tests based on the graph in terms of web page traversals [16]. The tool Veriweb explores sequences of links in web applications by nondeterministically exploring action sequences (*i.e.*, sequences of links) [1]. This tool provides data to forms using name-value pairs that provided by the tester.

Other testing techniques that attempt to test the effects of input values on web applications, but they require interface specifications and cannot guarantee code coverage without extensive user interaction. In some cases automated techniques derive the interface specifications [11] and in others developers must provide them [13], but either way, the testing system essentially performs fuzz testing that may be constrained by user-provided value specifications. Other testing mechanisms provide more reliable code coverage, but they repeatedly prompt the user for new inputs, so they sacrifice automation [18].

6.3 Static Analysis of PHP Web Applications

Static analysis of web applications is related to our work in the sense that it also attempts to find the same classes of bugs as our approach does. Jovanovich et al. developed Pixy as a taint-based analysis of PHP programs with constant propagation [14]. Xie and Aiken developed a similar analysis but sacrificed some precision in favor of scalability by using block and function summaries [30]. Because these analyses do not consider dynamically constructed string values, they can only check whether raw user inputs flow into sensitive sinks, although most injection vulnerabilities do fall into that category. Minamide constructed a string analysis that represents sets of string values using context-free grammars, although this is a more expensive analysis than standard taint analysis [19]. Wassermann and Su modified his analysis to add taint annotations to the generated grammars so that more expressive policies could be checked [29].

All of these techniques have limited effectiveness, because PHP supports dynamic features, in which the runtime system interprets data values as code, and dynamic features inhibit static analysis. The standard dynamic features PHP provides allow string values to specify: the name of a file to include, the name of a variable to read/write, the name of a method to invoke, the name of a class to instantiate, and the string representation of code to execute. All of the static analyses for PHP described above either fail on dynamic features, treat them optimistically (*i.e.*, ignore them), ask the user to provide a value for each one, or do some combination of the three. Many PHP applications use dynamic features extensively, for example, to implement dynamic dispatch for dynamically loaded modules or for database handling code. On such code, static analysis fails to produce useful results.

In most real-world PHP programs, however, the values of interpreted strings come only from trusted values such as constant strings within the PHP code, for example in a factory pattern; column names from a known database schema; or field names from a protected configuration file. In such cases, the values of interpreted strings depend only indirectly on user input, and for any given run, the predicates on user inputs are not dynamically constructed.

7. CONCLUSIONS

In this paper we have presented an approach for analyzing web applications by generating test inputs for them automatically using information from previous executions. This approach handles dynamic language features more gracefully than static analysis. We also explored new techniques in both the constraint generation and the constraint resolution phase. By projecting the trace onto a backward slice from the assertion of interest, we generated several orders of magnitude fewer constraints, and improved the scalability of the approach. By considering only one variable occurrence per expression, we improved the precision with which our constraint resolution algorithm models the semantics of operators and library functions.

In this paper, we have primarily considered a test oracle for SQL injection. In the future, we would like to explore other general test oracles. For example, in many web-based medical records systems, confidential information should not flow to certain classes of users. We are interested in exploring how multiple test executions can be compared to detect information leakage, and how inputs that will leak confidential information can be generated. The challenge will be in determining which predicates to target and in how new input values are selected.

We are also interested in exploring some implementation trade-offs further. For example, we would like to experiment with a constraint generation implementation that works as part of the runtime system, so that more can be done without tampering with the program. Part of the challenge will be in whether constraint generation can be done selectively but without the aide of a backward slice.

Acknowledgements

We thank Mark Gabel and the anonymous reviewers for their helpful comments on improving this paper.

8. REFERENCES

[1] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *Proceedings of the Eleventh International World Wide Web Conference (WWW 2002)*, 2002.

[2] T. S. BV. Tiobe programming community index, September 2007. URL: `http://www.tiobe.com/tpci.htm`.

[3] C. Cadar and D. R. Engler. Execution generated test cases: How to make system code crash itself. In *Model Checking Software, 12th International SPIN Workshop*, pages 2–23, 2005.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 322–335, 2006.

[5] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007 (SOSP 2007)*, pages 117–130, 2007.

[6] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software—Practice and Experience*, pages 1025–1050, 2004.

[7] E. de Vries, J. Gilbert, and P. Biggar. phc: The open source php compiler.

[8] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 151–162, 2007.

[9] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Proc. Black Hat USA*, 2007.

[10] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, pages 117–127, 2006.

[11] W. G. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2007)*, 2007.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley, Boston, MA, 2000.

[13] X. Jia and H. Liu. Rigorous and automatic testing of web applications, 2002.

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

[15] M. Kunc. What do we know about language equations? In *Developments in Language Theory, 11th International Conference (DLT 2007)*, pages 23–27, 2007.

[16] D. Kung, C. H. Liu, and P. Hsia. An object-oriented web test model for testing web applications. In *24th International Computer Software and Applications Conference (COMPSAC 2000)*, pages 537–542, 2000.

[17] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *16th International Symposium on Software Reliability Engineering (ISSRE 2005)*, pages 267–276, 2005.

[18] J. J. Li, D. Weiss, and H. Yee. Code-coverage guided prioritized test generation. *Information and Software Technology*, pages 1187–1198, 2006.

[19] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, 2005.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, 2007.

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

[22] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Object-Oriented Programming, 19th European Conference (ECOOP 2005)*, pages 504–527, 2005.

[23] W. Plandowski. Satisfiability of word equations with constants is in pspace. In *40th Annual Symposium on Foundations of Computer Science (FOCS 1999)*, pages 495–500, 1999.

[24] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[25] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 25–34, 2001.

[26] K. Sen and G. Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification, 18th International Conference (CAV 2006)*, pages 419–423, 2006. (Tool Paper).

[27] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2005)*, 2005.

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

[29] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 32–41, 2007.

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