Cross-Language Program Slicing for Dynamic Web Applications

Hung Viet Nguyen
ECpE Department
Iowa State University, USA

Christian Kästner
School of Computer Science
Carnegie Mellon University, USA

Tien N. Nguyen
ECpE Department
Iowa State University, USA

ABSTRACT
During software maintenance, program slicing is a useful technique to assist developers in understanding the impact of their changes. While different program-slicing techniques have been proposed for traditional software systems, program slicing for dynamic web applications is challenging since the client-side code is generated from the server-side code and data entities are referenced across different languages and are often embedded in string literals in the server-side program. To address these challenges, we introduce WebSlice, an approach to compute program slices across different languages for web applications. We first identify data-flow dependencies among data entities for PHP code based on symbolic execution. We also compute SQL queries and a conditional DOM that represents client-code variations and construct the data flows for embedded languages: SQL, HTML, and JavaScript. Next, we connect the data flows across different languages and across PHP pages. Finally, we compute a program slice for a given entity based on the established data flows. Running WebSlice on five real-world, open-source PHP systems, we found that, out of 40,670 program slices, 10% cross languages, 38% cross files, and 13% cross string fragments, demonstrating the potential benefit of tool support for cross-language program slicing in dynamic web applications.

Categories and Subject Descriptors
F.3.2 [Semantics of Programming Languages]: Program analysis

Keywords
Program slicing, dynamic web applications, cross-language analysis

1. INTRODUCTION
Program slicing [47] is an important and useful technique in several software engineering applications. For example, it is a useful tool to assist developers in understanding the impact of their changes for activities such as programming or bug fixing [3]. In general, a common way to estimate the impact of a change is to compute a program slice. A (forward) program slice for a variable \( C \) at some program point consists of all the parts of the program that may be affected by the value of \( C \) [47]. Thus, when a developer modifies some part of the program, performing slicing from the change point can reveal the potentially affected parts of the change.

While various program-slicing techniques have been developed for traditional software systems, program slicing for web applications is challenging due to their dynamic nature. The server-side code (often in PHP, ASP, JSP, etc.) dynamically generates HTML pages based on user input and data retrieved from databases. These pages often contain JavaScript (JS) code to enable interactive usage. The data from HTML forms can then be transferred back to the server side. When the server side receives the data, the generation process begins again with the new page.

The multilingual nature and the dynamic generation of client code in dynamic web applications raise challenges for program slicing. First, web applications are written in multiple languages, including server-side languages such as PHP and SQL, and client-side languages such as HTML and JS. Thus, data flows across different languages should be taken into account when computing a program slice. Second, client-side program entities (e.g., HTML input fields and JS variables) are often embedded in PHP string literals or computed via various string operations. A program-slicing technique would need to identify those embedded entities and recognize the data flows among them. Finally, the data flows for the embedded code might be governed by conditions in the server-side code. For example, the same PHP program may generate different HTML forms for different types of users; the data flows among the entities on these forms are dependent on the conditions in the PHP code.

This paper presents WebSlice, a technique to compute program slices for dynamic, multilingual web applications. We compute a program slice based on data-flow relations among entities (this type of slicing is called thin slicing [46]), including def-use relations (i.e., whether a reference refers to a definition of a variable) and information-flow relations [7] (i.e., whether a reference affects the value of a defined variable after executing a statement). We identify these relations for PHP code using an algorithm based on our symbolic-execution engine [36]. Symbolic execution also computes SQL queries and the output of the PHP program (possibly with symbolic values). To analyze the embedded code, we then parse this symbolic output with a variability-aware parser into a conditional DOM (called VarDOM) that represents all variations of the generated client-side code [14]. We analyze the SQL queries and the VarDOM to construct the data flows for each language of the embedded code: SQL, HTML, and JS. Next, we identify the data flows among data entities of different languages and across different PHP pages. Based on these established data flows, for a given data entity \( C \) at a point, we derive the program slice for \( C \) by including all the definitions and references that have direct or indirect data-flow relations with \( C \), possibly across different languages.
To investigate the necessity of program-slicing tool support in practical scenarios, we have run WebSlice on five real-world PHP systems. We found that out of 40,670 program slices, 10% cross languages, 38% cross files, and 13% cross strings. These results show that it may not be straightforward for developers to identify a slice manually without tool support. Our key contributions include:

1. An algorithm to build data-flow relations among data entities for server-side PHP and embedded code via symbolic execution,
2. An approach combining symbolic execution, variability-aware parsing, and data-flow analysis on embedded code to compute cross-language program slices for dynamic web applications, and
3. An empirical study to investigate the complexity of data flows and program slices in PHP web applications.

2. MOTIVATING EXAMPLE

To show the challenges in computing program slices for PHP web applications, we use a running example adapted from SchoolMate 1.5.4, an open-source web application for managing a school’s information (Figure 1a). The excerpt consists of two pages: ManageAnnouncements.php (Figure 1b) displays all announcements available in the database and allows a user to select one of them for editing. Upon selecting an announcement, the user is redirected to EditAnnouncements.php (Figure 1c) to update the details of the announcement. From there, the updated information is sent back to the first page, which updates the database of announcements with the new details. In this process, the data of the announcement is propagated across two stages (server side and client side), different languages, and multiple languages (PHP, HTML, SQL), as exemplified by the edges in Figure 1. Identifying such program slices is useful in a number of applications such as debugging and change impact analysis [19] (e.g., if developers make a change to encrypt the announcement ID, they can use the forward slice from the change point to investigate related parts that may be affected by the encryption).

Although desirable, program slicing for dynamic web applications faces a number of challenges:

1. Cross-language data flows: First, data entities can have relations across different languages and different PHP pages. That is, the value of a data entity computed in one language may affect the value of another entity in another language. For instance, on edge 4, the PHP variable $id is evaluated into a string and assigned to the value of the HTML input 'update' on the generated page. In addition, a data entity appearing in the execution of one page may refer to the value of another data entity appearing in a previous execution of another page (via different HTTP requests). For example, the PHP variable $POST['update'] (line 2 of Figure 1) refers to the value of the HTML input 'update' generated from a prior execution of another page.

2. Embedded entities: In a web application, there are different types of program entities written in multiple languages. In our program slicing analysis, we are interested in program entities that contain data such as PHP/JS variables and HTML forms and inputs. Let us call them (data) entities. Such entities appear in a program as definitions and references. A definition of an entity is the code location where the entity is declared or assigned with a value. A reference to an entity is the code location where the entity is referred to via the entity’s name.

Since the client-side code is dynamically generated from the server-side code, data entities can be embedded in PHP strings. That is, a definition/reference of an entity might be embedded within PHP string literals. As an example, the HTML input 'update' is concatenated from two PHP strings and a PHP variable (line 10 of Figure 1). HTML fragments from string literals can be printed directly through echo/print statements, but can also be assigned to PHP variables, propagated through computations, and printed out later. Thus, the relations between embedded entities could cross string literals and require an analysis on the semantics of embedded code.

3. Conditional client code: The data flows for embedded code might be determined by conditions in the server-side code (i.e., some of the dependencies are conditional). For instance, if different HTML forms are generated for different types of users (e.g., members or guests), the data flows and program slices for the entities in these forms are dependent on the conditions in the PHP code for those user types (not shown by our example).

3. WEB APPLICATION SLICING

In the literature, a (forward) program slice consists of the parts of a program that may be affected by the values computed at a slicing criterion, which is a point of interest typically specified by a program point and a set of variables [46]. Various program slicing methods have been proposed [47], since different properties of slices might be required for different applications. In this paper, we chose a class of program slicing that is based on data dependencies. This class is called thin slicing [46] as opposed to traditional slicing based on both data and control dependencies, which typically produces slices that are too large to be useful for human inspection. A full slice

Figure 1: A program slice in an example PHP web application

Data entity: [ ] Forward-slice edge: [ ]

Table 1: Slicing criteria for PHP web applications

<table>
<thead>
<tr>
<th>Slicing Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static slicing</td>
<td>Based on the static program structure</td>
</tr>
<tr>
<td>Dynamic slicing</td>
<td>Based on the dynamic program execution</td>
</tr>
<tr>
<td>Data-flow slicing</td>
<td>Based on the data-flow relations between entities</td>
</tr>
<tr>
<td>Control-flow slicing</td>
<td>Based on the control-flow relations between entities</td>
</tr>
</tbody>
</table>
Def-use  Def. to ref.  F1. A definition \(d\) and a reference \(r\) of a variable \(v\) have a def-use relation if there exists a control flow from the statement containing \(d\) to the statement containing \(r\) without interwining redefinitions of \(v\). For example, edge 3 in Figure 1 indicates a def-use relation between the definition and a reference of the PHP variable \(\$id\).

Info-flow  Ref. to def.  F2. For a statement \(S\), a reference \(r\) of variable \(v_1\) has an information-flow relation with a definition \(d\) of variable \(v_2\) if the value of \(v_1\) on entry to \(S\) may affect the value of \(v_2\) on exit from \(S\). As an example, in the statement \(\$x = \$y + \$z\), the references \(\$y\) and \(\$z\) have information-flow relations with the definition \(\$x\).

F3. A reference \(r\) and a definition \(d\) have a cross-language def-use relation if \(r\) and \(d\) are written in different languages, and \(r\) refers to the entity defined at \(d\). For instance, the PHP variable \$_POST[\'update\']\) refers to the value of an HTML input named \'update\' (edge 5 in Figure 1).

F4. A reference \(r\) in language \(L_1\) and a definition \(d\) in a different language \(L_2\) have a cross-language information-flow relation if \(r\) generates \(r^*\), and \(r^*\) forms the code that is used in the computation for the value of \(d\). For example, in the PHP code echo "<input name='input1' value='\$x'>", the value of the PHP variable \$x\) is assigned to the value of the HTML input 'input1'.

- **Table 1: Extension of data-flow relations for dynamic web applications**

<table>
<thead>
<tr>
<th>Relation</th>
<th>Direction</th>
<th>Within one language</th>
<th>Across languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Def-use</td>
<td>Def. to ref.</td>
<td>F1. A definition (d) and a reference (r) of a variable (v) have a def-use relation if there exists a control flow from the statement containing (d) to the statement containing (r) without interwining redefinitions of (v). For example, edge 3 in Figure 1 indicates a def-use relation between the definition and a reference of the PHP variable ($id).</td>
<td>F3. A reference (r) and a definition (d) have a cross-language def-use relation if (r) and (d) are written in different languages, and (r) refers to the entity defined at (d). For instance, the PHP variable $_POST['update']) refers to the value of an HTML input named 'update' (edge 5 in Figure 1).</td>
</tr>
<tr>
<td>Info-flow</td>
<td>Ref. to def.</td>
<td>F2. For a statement (S), a reference (r) of variable (v_1) has an information-flow relation with a definition (d) of variable (v_2) if the value of (v_1) on entry to (S) may affect the value of (v_2) on exit from (S). As an example, in the statement ($x = $y + $z), the references ($y) and ($z) have information-flow relations with the definition ($x).</td>
<td>F4. A reference (r) in language (L_1) and a definition (d) in a different language (L_2) have a cross-language information-flow relation if (r) generates (r^<em>), and (r^</em>) forms the code that is used in the computation for the value of (d). For example, in the PHP code echo &quot;&lt;input name='input1' value='\$x'&gt;&quot;, the value of the PHP variable $x) is assigned to the value of the HTML input 'input1'.</td>
</tr>
</tbody>
</table>

4. **APPROACH OVERVIEW**

We propose WebSlice, an approach to compute program slices in a PHP web application. WebSlice proceeds in four main steps: (1) performing symbolic execution on the PHP code to approximate its output as well as constructing the data flows for server-side code in PHP and SQL, (2) parsing and analyzing the output to construct the data flows for client-side code in HTML and JS, (3) connecting the data flows across different languages, and (4) computing a slice given a slicing criterion. Figure 2 gives an overview of these steps.

---

**Step 1—Symbolic execution for data-flow analysis:** The goal is two-fold: (1) to approximate the output of a PHP program so that the data flows within embedded client code can be analyzed in later steps and (2) to construct the data flows within the server-side code. For approximating the output, we reuse our symbolic-execution engine [15]. The result of symbolic execution is the generated client-side code which possibly contains symbolic values and values that are produced under specific path constraints. For illustration, the output of the code in Figure 1 is shown below, with Greek letters for symbolic values and \# if directives (similar to those in C preprocessing) representing texts that are output under some constraints.

1. \(#\) if \(\alpha / \$_POST[\'edit\'] \) == 1
2. \(<div>\)Database updated.(</div>)
3. \(#\) endif
4. ... \(<form name='announcements' ... >
5. \(#\) if \(\beta / \) mysql_fetch_array (PreparedStatement)
6. \(<input ... name='update' value='\$x'>\) /\(\theta\) represents \$id
7. \(#\) endif
8. \(<input type='submit' value='Edit'></form>..."
by checking the satisfiability of the path constraints under which the
data entities appear.

```plaintext
1 if ($_GET['user'] == 'admin')
2   $message = 'Welcome admin!';
3 else
4   $message = 'Access denied!';
5 if ($_GET['user'] == 'admin')
6   echo '<div class="msg-admin">'. $message .'</div>'; // Fig. 1
```

For example, in the code above, the PHP variable definition $message
on line 4 does not have a def-use relation with the reference on
line 6 since they are under different path constraints. Also, symbolic
execution allows us to resolve dynamically included PHP files,
thereby detecting data flows that would otherwise be missed.
For scalability, we have made several approximations to our symbolic
executor such as running at most two iterations of a loop and skipping
recursive function calls; we discuss them in Section 5.

Step 2—Embedded code analysis: To detect data flows within
embedded client code, we first parse the output produced from the
previous step with our HTML and JS variability-aware parsers [5].
The parsers were developed using the TypeChef variability-aware
parser framework [29], which enables parsing programs with condi-
tional parts (e.g., an HTML opening tag may have two different
closing tags depending on a constraint). Note that variability-aware
parsing is sound and complete w.r.t. brute-force parsing of all possi-
ble variants, but much faster [25] [16]. The parsing result is a
VarDOM representation [34] of the embedded client code, which
represents the hierarchical DOM structure of a web page, as shown in
Figure 3 for the example in Figure 1. Unlike an HTML DOM, a
VarDOM contains condition nodes to indicate that certain VarDOM
entities can be conditional (i.e., some parts of the web page are
produced under some path constraints). For example, as seen in
Figure 3 the HTML input field ‘update’ is displayed under con-
straint β (mysql_fetch_array($query)), and its value attribute has a
symbolic value θ (Sid). In this way, the VarDOM represents all
possible generated client pages (with symbolic values) in a single
tree structure. A VarDOM also contains conditional AST(s) for JS
code with conditional and symbolic parts (not shown in Figure 3).

Using the VarDOM representation of the client code, we are able
to analyze the embedded code written in HTML/JS and build their
respective data flows. Since HTML is a declarative language, we
collect the definitions of HTML entities. For JS code, we compute
its data flows using a light-weight symbolic-execution engine with
an algorithm to build data flows similar to that for PHP.

Figure 3: The VarDOM representation for the output of the program
in Figure 1

Step 3—Connecting data flows: As seen in Figure 1 data flows
can exist among data entities of different languages and across
different pages. Thus, we connect the data flows among those entities
based on cross-language def-use and information-flow relations (F3
and F4 in Table 1). For instance, the input fields in the HTML form
<form action = 'EditAnnouncements.php' ...
> have cross-language
def-use relations with the corresponding PHP $_GET/$POST vari-
bles on the page ‘EditAnnouncements.php’ since the inputs field are
submitted to that page. Through this step, we obtain the data flows
for the entire web application. Figure 4 shows the data-flow graph
for our example, in which the nodes represent the definitions/refer-
ces of entities and the edges represent (direct) data-flow relations.

Step 4—Computing slice: Once the data-flow graph is produced,
we can use it to quickly compute any program slice. Given a def-
dition or a reference C, the program slice for C consists of the
definitions and references that are reachable from C in the graph.

5. DATA-FLOW ANALYSIS

This section presents our algorithm to construct the data flows and
to compute the output and SQL queries of PHP code. The algorithm
is built on top of our symbolic-execution engine [56]. We first
introduce the notation that will be used to describe our technique.

5.1 Notation

V is the set of all values (including symbolic ones). C is the set of all
control codes that represents the returned values of statements (e.g.,
‘RETURN’ or ‘BREAK’). S, E, and N are the sets of all statements,
expressions, and identifiers, respectively. Π is the set of all path
constraints; each constraint is a propositional formula. L is the set
of all definitions and references. We use small letters for elements
of a set (e.g., s ∈ S is a statement).

Our symbolic-execution engine processes a PHP program and
considers all unknown values, such as user input and data from a
database, as symbolic values. When reaching a control predicate,
it explores feasible paths and keeps track of the path constraint
for each path. Specifically, we maintain a program state (V, D, Π)
where the value store V: N → V is a (total) function mapping a
variable/function name to its value (uninitialized variables have a
value). The path constraint Π encodes the branch decisions taken
to reach the current state. In addition to V and Π, which are typical
for symbolic executors, to detect data flows, we track a definition
store D: N → Π(L × Π) that maps each variable name to its set
of definitions together with a path constraint under which each
definition appears.
Figure 5: Excerpt of PHP symbolic execution’s evaluation rules (extensions to PhpSync [36] are highlighted in gray)
Our symbolic executor evaluates statements in a PHP page and repeats this process for other pages to build the data flows of the entire web application. Figure 6 shows the key evaluation rules. For a statement $s$, a rule \( (s, \mathcal{V}, \mathcal{D}, \pi) \rightarrow (e', \mathcal{V}', \mathcal{D}', \pi') \) denotes that the execution of $s$ changes the program state from $(\mathcal{V}, \mathcal{D}, \pi)$ to $(\mathcal{V}', \mathcal{D}', \pi')$. The returned value $e$ is a control code: It returns `OK` if there was no control-flow breaking instruction in $s$ (i.e., the next sequential statement can be executed) and other control codes (e.g., `RET` or `RETURN`) otherwise. For an expression $e$, a rule \( (e, \mathcal{V}, \mathcal{D}, \pi) \rightarrow (v, \mathcal{V}', \mathcal{D}', \pi') \) denotes that the evaluation of $e$ results in a new program state and returns a (non-control) value $v$.

In Figure 5, we formalize the rules for our previous execution engine and highlight the parts that we extend to identify data-flow relations. \( \text{addEntity} \) and \( \text{addRelation} \) are used to create the nodes and edges of the data-flow graph (the graph is a global data structure and is not shown in the program state). We use \( \text{addOutput} \) to record a string or symbolic value in the output (under path constraint). Other notation and auxiliary functions are listed in Figure 5.

### 5.2 Intraprocedural Data Flows (Rules 1–3)

During symbolic execution, we detect data flows by identifying def-use and information-flow relations among data entities. For def-use relations, since a reference could have multiple definitions (e.g., a PHP variable can be defined in different branches and then later accessed after the branches), we need to keep track of the set of definitions of each reference. Therefore, we maintain these sets via the definition store $\mathcal{D}$. When a reference $r$ with name $n$ is found under a path constraint $\pi$, we look up its definitions in the set $\mathcal{D}(n)$ and match $\pi$, with the constraints of those definitions to retain only feasible relations. Specifically, a definition $d$ with constraint $\pi_d$ in $\mathcal{D}(n)$ has a feasible def-use relation with $r$ if $\pi_d \land \pi$ is satisfiable (i.e., there exists at least one execution path where both $d$ and $r$ appear), as shown in rule 1 of Figure 5.

To identify information-flow relations, at a variable assignment, we record the information flow from the variables on the right-hand side to the one defined on the left-hand side (rule 2). Note that if the right-hand side of an assignment contains a user-defined function call, the arguments in the function call do not have direct information-flow relations with the defined variable; we detect their relations through interprocedural data flows instead (Section 5.3). We also update the definition store $\mathcal{D}$ with the new definition of the variable. If a variable is redefined through sequential statements, we overwrite its previous definitions with the new definition since values from the previous definitions can no longer be accessed. If a variable is defined/undefined in branches of a conditional statement, we keep the values/definitions of the variable independent in the branches but combine them after executing all branches. Let us describe the details next.

### 5.3 Interprocedural Data Flows (Rules 4–6)

Similar to concrete execution, symbolic execution evaluates a function call in three steps (if the source code is available; otherwise, it returns a symbolic value): First, it sets up a new context/call stack for the function and passes the actual parameters to the formal parameters of the function. Second, it executes the function body and records all returned values (here represented by a special variable named `RET`) encountered when exploring different paths in the function. Third, in the returned value(s) are propagated to the call site of the function. Our extended algorithm instruments this process and tracks the data flows: We create a definition for each formal parameter and record the data flows from the arguments of the function call to the parameter definitions in the function declaration. To track the data flow from the function to its call site, we create two special RET nodes: a RET$_{def}$ node representing the return value computed inside the function and a RET$_{ref}$ node representing the

---

**Figure 6:** Detecting data flows at conditional statements

**Figure 7:** Interprocedural flows with highlighted RET nodes

Handling conditional statements (rule 3): If the path constraints of both branches of an if statement are satisfiable, we explore both branches. The function whenEqual($v$, TRUE) is used to compute the constraint where a value $v$ (evaluated from the if condition) evaluates to TRUE. For example, the conditions on lines 2 and 7 of Figure 6 are both resolved into $\alpha = 'admin'$ where $\alpha$ is the symbolic value for $_GET['user']$; thus, we evaluate the boolean expressions in the (fresh) symbolic value $\mu$ (to simplify constraint checking). Modifications to the definition store $\mathcal{D}$ (as well as the value store $\mathcal{V}$) take effect in the corresponding branch only. After executing the branches, we update the definition store of the (value store) with the combined definitions and values from the two branches and their corresponding constraints. To represent that a variable may have multiple values depending on a path constraint, we use an \( \text{ite}(\pi, v_1, v_2) \) value (short for \( \text{if-then-else} \)) to denote a selection between value $v_1$ if the path constraint $\pi$ evaluates to TRUE and $v_2$ otherwise (we also use \( \text{ite} \) for control codes). Note that if the path constraint of one of the branches is unsatisfiable, we execute the other (satisfiable) branch only. (In rule 3, we show one such case; the other case is symmetric.) As an illustration, in Figure 6, the variable $\$message$ after line 6 has two definitions from both branches. When the variable is accessed under constraint $\mu$ on line 8, we compare its constraint with the constraints of the definitions in $\mathcal{D}$ to eliminate an infeasible relation with the definition on line 5.
propagated return value at the call site. Note that if a function is invoked multiple times, we create separate entities, RET nodes, and data flows corresponding to each function invocation (for each invocation, the execution path in the function body could be different depending on the specific input arguments). Since we create different contexts at function calls, the approach does not suffer from the calling-context problem \[7\], caused by analyzing different function calls in the same context, which would result in infeasible interprocedural data flows. To illustrate, Figure\[7\] (right-hand side) shows the interprocedural data flows for the PHP variable \$welcome (line 7) and \$login (line 9). In the code, we show the data flow for \$welcome only; the data flow for \$login is similar. Note that one code location may correspond to several nodes in different contexts (e.g., the two nodes labeled L1: \$content since the createDiv function is executed twice. The detailed rules are shown in rules 4–6.

5.4 Handling Special Statements (Rules 7–9)

Handling a block of statements (rule 7): In a block of statements, the returned control code after executing a statement can be ‘OK’, indicating that the next statement can be executed, or other control codes otherwise (e.g., ‘RETURN’ for a return statement). Note that the returned control code can also be an ite code (the returned code of an if statement). Therefore, after each statement, we compute the path constraint under which the next statements can be executed (i.e., the constraint with which the returned control code equals ‘OK’) and execute them under that restricted constraint. After executing the block, we update the definition store and the value store similarly to the case of an if statement. If the computed constraint is not satisfiable, we simply stop the execution for that block (not shown). In rule 7, we show the algorithm for a block of two statements; the rule for a block with more statements can be generalized from this rule.

Handling loops (rule 8): For a loop, we aim to detect data flows across different iterations. For instance, in the code snippet below, there is a def-use relation from the variable \$y on line 3 to the variable \$y on line 2 if the loop can be executed multiple times.

```
1 while ( \$row = mysql_fetch_array( \$result )) {
2     \$x = \$x + 1;
3     \$x = \$x * 2;
4 }
```

Therefore, to detect such data flows, we execute the body of a loop at most twice by modeling the loop as two nested if statements and applying the rule for if. If the loop contains control-flow breaking instructions (such as break, continue, or exit), we either abort the loop (for break, return, and exit) or continue the next iteration (for continue) in their respective constraints (not shown).

Handling dynamically included files (rule 9): A PHP program can dynamically include other files. During symbolic execution, we execute these files if the file names can be resolved to concrete values. Since include is an expression in PHP, we treat the returned value of include similarly to the returned value of a function call.

Handling aliasing and objects (rule not shown): When a PHP object is created, we maintain two maps from the object’s fields to their values and definitions (similar to the stores \(V\) and \(D\)). Therefore, even if an object field is written and read via different variables (through aliasing), our algorithm can still recognize a def-use relation between the definition and reference, as illustrated below. (The same mechanism is used to handle assignment/call by reference.)

5.5 Approximating the Output (Rules 10–11)
The output of a PHP program is usually a concatenation of multiple string values and is printed out through echo/print statements or inline HTML code. To keep track of concatenations, we use a concat \((v_1, v_2)\) value representing a concatenation of two (possibly symbolic) values \(v_1\) and \(v_2\) (rule 10). At echo/print statements or inline HTML code, we simply record the computed value \(v\) for the output in the corresponding path constraint (rule 11). The use of concat values (together with \(ite\) values) allows us to track the symbolic output with conditional fragments precisely and compactly, making the subsequent variability-aware parsing on the output efficient while preserving path constraints.

5.6 Data Flows between PHP and SQL (Rules 12–14)
In a web application, to retrieve data from a database, one can construct an SQL query and invoke PHP functions for database queries such as mysql_query. The returned data is stored in a record set with rows and columns. To iterate through each row in the record set, a PHP function such as mysql_fetch_array can be used. To access each column in a row, one can access the corresponding column name/index of the array containing the row. Since such an array access in PHP retrieves data originating from a database, we consider it as a data flow (def-use relation) from SQL to PHP. In that def-use relation, we consider the SQL table column name appearing in the SQL query as an SQL definition and the corresponding array access as a PHP reference to an SQL entity. For instance, on line 7 of Figure\[4\], \$bulletinid is an SQL table column definition, having a def-use relation with the PHP array access on line 9.

To detect such data flows, during symbolic execution, we input the value of an SQL SELECT query, which could also contain symbolic/conditional characters, into our variability-aware SQL parser (similar to the HTML parser in Section 6) to recognize table column names as SQL definitions (function parseAndFindSqlDefs in rule 12 of Figure\[5\]). This set of SQL definitions is propagated through mysql_fetch_array function calls (rule 13). When there is an array access to such SQL data, we detect a relation between them (rule 14). In this work, we detect data flows from SQL to PHP; we plan to apply similar ideas for data flows within SQL and from PHP to SQL (via SQL INSERT/UPDATE statements).

5.7 Traditional vs. Thin Slicing
We compute a thin slice by including all reachable nodes from a given node in the data-flow graph. However, we could easily record control dependencies for traditional slicing as follows. At an if statement (rule 3), we could additionally record the control dependencies between references on the if condition and the definitions within its branches and extend our graph to have both control and data dependencies on entities (similar to a PDG on statements). We can then reduce program slicing to a reachability problem on this graph.

Limitations: Currently, our symbolic-execution engine handles the common but not all PHP constructs. For instance, we implement infix expressions with the concatenation operator only, since we are interested in the string output of a program. For other operators, we create fresh symbolic values. For instance, we track \(\alpha > 1\) as a new symbolic value \(\beta\); therefore, we may explore some infeasible paths. The executor runs at most two iterations of each loop and skips recursive function calls. We discussed these simplifications and why they are acceptable for approximating the program’s output elsewhere \[36\]. Regarding data-flow detection, if an array access cannot be resolved, we track data flow from the array variable instead (e.g., edges 7 and 8 in Figure\[4\]).
6. EMBEDDED CODE ANALYSIS

We parse the symbolic output of a PHP program with our HTML and JS variability-aware parsers [34] into a VarDOM representation of the client-side code (Figure 3). We then analyze the VarDOM to collect data entities and construct data flows for the embedded code.

Analyzing HTML: Since HTML is a declarative language, we detect the definitions of HTML entities by traversing the VarDOM tree and identifying the following types:

1. **HTML definitions by name**: These entities are identified by the ‘name’ attribute of an HTML element (e.g., `<form name="form1">`).
2. **HTML definitions by ID**: These entities are identified by the ‘id’ attribute of an HTML element (e.g., `<div id="id1">`).
3. **HTML definitions by URL parameters**: These entities are detected in HTML query strings (e.g., the data entity lang in `<a href="#lang=en">`).

Building data flows for JS: To construct data flows for JS, we first extract JS code from JS locations on the VarDOM. These locations include HTML `<script>` tags and HTML event handlers (e.g., onload, onclick). The VarDOM already contains the parsed JS ASTs for these code fragments [34], each of which serves as an entry point. We then use a light-weight symbolic-execution engine for JS that is similar to the one for PHP (by adapting the rules in Figure 5 for JS), run it for every entry point and detect data flows. Currently, we do not handle client code that is dynamically generated from JS code such as `document.write` or `eval`, and data flows involving AJAX.

7. CROSS-LANGUAGE DATA FLOWS

Data flows can exist among entities of different languages (F3 and F4 in Table 1). In Figure 3, we show all possible def-use and information-flow relations across languages. We detect these cross-language flows as follows.

**F3—Cross-language def-use relations**: Table 2 shows the types of cross-language def-use relations in a web application.

1. **Between HTML/JS and PHP (rows 1–3)**: A PHP program can access data sent from a client page via PHP `$_POST` or `$_GET` arrays (corresponding to HTTP POST/GET protocols or both). These arrays hold key/value pairs, where the keys are the names of the HTML input fields. Therefore, we identify those array accesses as `PHP references to client-side entities`. Note that the submitted destination of the client-side entities (specified by the ‘action’ attribute of an HTML form or the address part in a URL) must match the PHP page containing the PHP reference.

2. **Between SQL and PHP (row 4)**: As described in Section 5.6, we detect these relations during our symbolic execution on PHP.

3. **Between HTML and JS (rows 5–8)**: In the client code, JS can operate on HTML elements via the HTML DOM. For example, the JS expression `document.form1.input1.value` retrieves the value of an HTML input field named ‘input1’ in a form named ‘form1’.

We identify these JS expressions as `JS references to HTML entities`. However, if they appear on the left-hand side of an assignment, we consider them as `JS definitions of HTML entities` instead since they redefine the values of the corresponding HTML entities. Similar to detecting data flows in PHP, we also check the path constraints under which these client-side entities are generated to eliminate infeasible data flows among them.

**F4—Cross-language information-flow relation**: During symbolic execution on PHP or JS, we track any generated string value (or symbolic value) to the variable or expression that generates it. If the value is used in an information-flow relation in the generated code, we recognize it as a cross-language information-flow relation from the generating language. For example, the value of the HTML input field ‘update’ in Section 2 is a symbolic value θ (Figure 3). During symbolic execution, we track θ to the PHP variable $id. Thus, we detect a cross-language information-flow relation between the variable $id and the HTML input’s value (see edge 4 in Figure 1).

We apply the above process for a (predefined) set of page entries (PHP files that can be requested by a web browser) to build the data flows within individual pages (the data flows for a page can involve multiple files). To detect data flows across page entries, we detect types 1–3 in Table 1 (The other types in Table 1 are applicable for within-page relations only.) Data flows via cookies and sessions are currently not supported. Note that the resulting data-flow graph may contain identical clusters of nodes where there are no edges across those clusters and the clusters all correspond to the same code locations in the server-side program (since the same code might be executed multiple times); in such cases, we retain only one cluster and discard the others.

Calling-context problem with inter-page data flows: When a client page submits data to the server side, the corresponding server-side program is invoked to handle the request. Conceptually, this process is similar to invoking a function call from the client page in which the arguments to the function call are the client’s data. Although we could handle the invocation of pages similarly to function calls, in our current implementation, we do not execute a page entry multiple times. Thus, the calling-context problem may occur for inter-page data flows, resulting in some infeasible data flows. However, our test results on a real-world system indicated that this problem does not cause significant imprecision.

8. IMPLEMENTATION

We implemented our WebSlice approach as an Eclipse plugin. WebSlice extends our previous symbolic-execution engine [35] and variability-aware parsers [34]. We use TypeChef’s library for propositional formulas [25] with a JavaBDD backend [2] for tracking path constraints and checking satisfiability.
Program slicing tools are intended to help developers in various software maintenance tasks such as identifying the impact of a change. To evaluate a slicing approach, one could design a user study to show how often such complicated cases occur. Therefore, we designed a study to quantify characteristics of data-flow dependencies and slices in existing web applications. Specifically, we are interested in how many entities are embedded within PHP strings, how many data-flow edges are cross-language or cross-string, how many slices cross languages and even web pages or require investigating embedded code fragments—all properties for which no current slicing tool is available. Although such complexity measures are only proxies for actual developer tasks, we argue that identifying a large set of complex data-flow dependencies or slices would demonstrate the benefit of automated slicing in dynamic web applications.

Table 4: Complexity of data-flow graph

<table>
<thead>
<tr>
<th>System</th>
<th>PHP entities Total</th>
<th>Non-PHP entities Total</th>
<th>Data-flow edges Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>10,591</td>
<td>266</td>
<td>13,406</td>
</tr>
<tr>
<td>SM</td>
<td>4,935</td>
<td>2,042</td>
<td>6,945</td>
</tr>
<tr>
<td>TC</td>
<td>15,291</td>
<td>2,145</td>
<td>16,490</td>
</tr>
<tr>
<td>UPB</td>
<td>32,309</td>
<td>1,308</td>
<td>36,186</td>
</tr>
<tr>
<td>WC</td>
<td>3,805</td>
<td>497</td>
<td>3,934</td>
</tr>
<tr>
<td>Total</td>
<td>66,931</td>
<td>6,618</td>
<td>76,961</td>
</tr>
</tbody>
</table>

N-Echo: Embedded entities that are not on echo/print statements; xLang, xFile, xFunc, xString, xPage: Edges that cross languages, files, functions, strings, and page entries.

To test the resulting data-flow graphs (and program slices) computed by WebSlice, we created 100 test cases for SchoolMate-1.5.4, a real-world web application that we used in our study in Section 9, covering all types of data flows. For data flows within PHP, we instrumented Quercus [1], an existing PHP interpreter, and dynamically tracked actual data-flow relations as a basis for our test oracles. For cross-language and cross-stage data flows and those within JS, we created the test cases manually. There are 20 test cases that include inter-page data flows; 2 of them failed because WebSlice included an infeasible edge (see last paragraph in Section 7). All test cases for other types of data-flow edges passed. More information about WebSlice can be viewed on our website [1].

9. EMPIRICAL STUDY

To answer those questions, we collected from sourceforge.net five PHP web applications with various sizes (Table 3), a corpus also used in prior and related work [34, 42]. For each system, we automatically chose a set of page entries (i.e., PHP files that generate output containing an <html> tag) and ran WebSlice on those pages to create the data-flow graph for the entire system. To compute the slices, we considered each entity (a node in the data-flow graph) as a slicing criterion and calculated the program slice for the entity.

Table 5: Complexity of slices

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Slices</th>
<th>Size</th>
<th>Len</th>
<th>xLang</th>
<th>xFile</th>
<th>xFunc</th>
<th>xString</th>
<th>xPage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>6,827</td>
<td>6</td>
<td>5</td>
<td>287</td>
<td>2,330</td>
<td>3,202</td>
<td>344</td>
<td>243</td>
</tr>
<tr>
<td>SM</td>
<td>4,185</td>
<td>19</td>
<td>4</td>
<td>1,518</td>
<td>1,519</td>
<td>917</td>
<td>1,735</td>
<td>890</td>
</tr>
<tr>
<td>TC</td>
<td>9,145</td>
<td>4</td>
<td>1,193</td>
<td>2,007</td>
<td>643</td>
<td>1,378</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>UPB</td>
<td>17,906</td>
<td>7</td>
<td>5</td>
<td>795</td>
<td>7,904</td>
<td>8,386</td>
<td>1,236</td>
<td>681</td>
</tr>
<tr>
<td>WC</td>
<td>2,607</td>
<td>4</td>
<td>3</td>
<td>312</td>
<td>1,557</td>
<td>1,517</td>
<td>408</td>
<td>265</td>
</tr>
<tr>
<td>Total</td>
<td>40,670</td>
<td>5.6</td>
<td>4.2</td>
<td>4,105</td>
<td>15,317</td>
<td>14,665</td>
<td>5,101</td>
<td>2,303</td>
</tr>
</tbody>
</table>

Slices that cross languages, files, functions, strings, and page entries.

9.1 Experiment Setup

To test the resulting data-flow graphs (and program slices) computed by WebSlice, we created 100 test cases for SchoolMate-1.5.4, a real-world web application that we used in our study in Section 9, covering all types of data flows. For data flows within PHP, we instrumented Quercus [1], an existing PHP interpreter, and dynamically tracked actual data-flow relations as a basis for our test oracles. For cross-language and cross-stage data flows and those within JS, we created the test cases manually. There are 20 test cases that include inter-page data flows; 2 of them failed because WebSlice included an infeasible edge (see last paragraph in Section 7). All test cases for other types of data-flow edges passed. More information about WebSlice can be viewed on our website [1].

9.2 Complexity of Data-Flow Graphs

We also used our tool to investigate the complexity of data-flow graphs in dynamic web applications. Table 4 shows the complexity of data-flow graphs. Overall, developers would have to deal with a large number of SQL, HTML, and JS entities. There are a total of 4,203 HTML entities in all five systems, accounting for 64% of all non-PHP entities. There exist cases where developers would deal with up to 384 HTML entities in a file (e.g., in TimeClock) and up to 88 JS entities in a file (e.g., in SchoolMate). Especially, they must process as many as 89% of the non-PHP entities by examining embedded code in PHP strings (the remaining are directly embedded in PHP code). Moreover, not all embedded entities are printed directly on echo/print statements: 18% of them are assigned to variables, propagated through the program, and printed out at a different location, which makes it challenging to track the data flows without tool support. The edges in the data-flow graphs also demonstrate significant complexity. Out of 76,961 data-flow edges, 12% cross languages, 21% cross files, and 14% cross strings. This result shows that tool support would be useful in those cases.
9.3 Complexity of Program Slices

Table 5 shows complexity metrics for slices (we exclude those that have only one entity since the entity is at the end of data flows).

Size of a slice: We compute the medians of sizes and lengths of slices and calculate their averages. On average, a developer would need to deal with a slice involving 5.6 entities and having a length of 4.2 (the longest path in the data-flow graph starting from a slicing criterion). 10% of the slices involve more than 40 entities (not shown in Table 5), which would be nontrivial to identify manually.

Cross-language data flows in a slice: Importantly, many of the slices are cross-language (in all five systems, 4,105 slices contain at least one cross-language data-flow edge). As shown in Figure 9, 35.8% of those slices have at least 4 cross-language data-flow edges, and 9.5% have at least 32 cross-language edges.

Cross-location data flows in a slice: Many slices are also often cross-location: 38% of the slices cross files, 36% cross functions, and 13% cross string fragments.

9.4 Discussion

Implications: The high complexity of the data-flow graphs and program slices shows that in real-world web applications, manually inspecting a program slice can be challenging, and developers would likely benefit from program slicing tool support.

Performance: The initial symbolic execution on all entries and construction of the data-flow graphs completed within a few seconds/entry for all systems. When the source code is changed, WebSlice needs to re-compute relevant page entries associated with the change only. This means that WebSlice can be run in the background of an IDE. Once the initial computations are finished, WebSlice can instantly show the program slice for any selected program point.

Threats to Validity: Regarding external validity, we used only a small set of medium-sized subject systems due to our limited support for PHP object-oriented constructs. Regarding construct validity, we used complexity as a proxy metric to show the usefulness of our program-slicing technique.

10. RELATED WORK

There exist excellent surveys on techniques for program slicing [47, 5, 18, 19, 28, 30, 10, 51]. Harman et al. [18] provide an extensive survey with multiple dimensions to classify program-slicing techniques. Later, Silva extends the dimensions [45]. We compare WebSlice with the related work in the context of those dimensions.

To approximate the dynamically generated client code, Tonella and Ricca [48, 40, 41] propose a flow analysis called string-cat propagation to associate the variables used in print/echo statements to string concatenations. They also combine with code extraction, which unquotes the strings in echo. The slice is computed from such flows. In contrast, our symbolic execution with variability-aware analysis is applicable to general cases and a wide range of PHP constructs. We also handle conditional flows and embedded SQL/JS code. Unfortunately, according to our correspondence with the authors, their tool and data are no longer available for comparison.

WebSlice is close to the information-flow approach by Bergeretti and Carré (BC) [7]. Regarding the algorithms, the information-flow relations in BC are recursively computed in a syntax-directed, bottom-up manner. We use symbolic execution to detect the flow relations. Regarding Silva [45]'s and Harman et al. [18]'s dimensions, WebSlice has key differences. First, we consider more relations to compute the slices, e.g., cross-language def-use and information-flow relations (Table 1). Second, regarding path-awareness dimension, unlike BC, WebSlice is path-sensitive but unsound. Finally, for dimension of iteration counts, we symbolically execute each loop twice to detect cyclic data flows. BC does not handle cyclic flows.

There are static slicing approaches based on various static analyses, e.g., incremental slicing [13], call-mark slicing [17], proposition-based slicing [22], stop-list slicing [13], amorphous slicing [17]. WebSlice is related to PDG-based slicing [39, 23]. However, we must deal with flows to embedded code. There are dynamic slicing approaches [1, 21, 22, 24, 29], including language-independent slicing [9], which compute a slice for one specific execution whereas WebSlice produces a static slice for all possible executions.

WebSlice differs from the family of conditioned program slicing [11, 12], constraint slicing [14], and pre/post-conditioned slicing [21], where an initial state is defined via conditions.

WebSlice differs from the family of conditioned program slicing [11, 12], constraint slicing [14], and pre/post-conditioned slicing [21], where an initial state is defined via conditions.

There exist string analysis approaches for web programs and software security [29, 25, 30, 52, 31]. They can be used to extract embedded code in our analysis. Maute et al. [30] and Ngo and Tan [33] extract database interactions, whereas WebSlice extracts only database columns’ names and slices through PHP.

In our prior work, we designed a simpler symbolic execution engine [36] to approximate PHP code’s output. Subsequently, we developed DRC [35] to analyze both PHP and client-side code to detect embedded dangling references. Later, we built HTML/JS variability-aware parsers in Varis [34], using TypeChef [25] to produce the VarDOM, a representation of all possible variants of client-side code. We built data flows and slices upon them as explained. Our symbolic execution on JS is similar but simpler than the one for PHP. We could also explore Kudzu [43], a powerful engine for JS.

11. CONCLUSION

In this paper, we tackled the challenge of computing program slices with multiple languages for dynamic web applications. We introduced WebSlice, an approach that combines symbolic execution, variability-aware parsing, and data-flow analysis on embedded code to identify PHP data entities as well as embedded SQL/HTML. JS entities and recognize the data flows among them within one language and across different languages. In our empirical study on five real-world PHP systems, we found many nontrivial cross-language cross-string program slices. In such cases, cross-language program slicing tool support such as WebSlice could be useful in assisting web developers with software maintenance tasks.

12. ACKNOWLEDGMENTS

This project is funded in part by National Science Foundation grants: CCF-1318808, CCF-1018600, CNS-1223828, CCF-1349153, CCF-1320578, and CCF-1413927.
13. REFERENCES

http://home.engineering.iastate.edu/~hungnv/Research/WebSlice


[3] Quercus interpreter for PHP. 
http://quercus.caucho.com/


