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 those 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 [5]. 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 [34]. 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.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2786805.2786872

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 1). The excerpt consists of two pages: ManageAnnoucements.php (Figure 1a) 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 1b) 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 pages, 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 1b) 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 1a). HTML fragments from string literals can be printed

a) ManageAnnouncements.php

```
1  <?php
2  if ($_POST['edit'] == 1) {
3  mysql_query("UPDATE bulletins SET message=`$_POST['message']`
           WHERE sbulletinid = $_POST['announcementid']");
4      print ("<div>Database updated.</div>");
5  } ...
6  print ("<form name='announcements' action='EditAnnouncements.
         php' method='POST'>");
7  $query=mysql_query("SELECT sbulletinid,message FROM bulletins..")
8  while ($announcement = mysql_fetch_array($query)) {
9      $id = $announcement[0];
10 print ("<input name='update' value ='" . $id . "'/>...");
11 }
12 print ("<input type='submit' value='Edit'>
13         </form>...");
14 ?>
```

b) EditAnnouncements.php

```
1  <?php
2  $id = $_POST['update'];
3  $query = mysql_query("SELECT * FROM bulletins
           WHERE sbulletinid =$id[0]");
4  $announcement = mysql_fetch_array($query);
5  print ("<form name='editform' action='ManageAnnoucements.php'
              method='POST'>
6   <input type='text' name='message' value='$announcement[1]' />
7   <input type='button' value='Edit' onclick='checkAndSubmit()' />
8   <input type='hidden' name='edit' />
9   <input type='hidden' name='announcementid' value='".$id[0]."' />
10 </form>");
11 ?>
12 <script> function checkAndSubmit() {
13     if (document.editform.message.value != '') {
14         document.editform.edit.value = 1;
15         document.editform.submit();
16     }
17 } </script>
```

Data entity: Forward-slice edge:

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

directly with 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 [47]. 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

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

Relation	Direction	Within one language	Across languages
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 intervening redefinitions of v. For example, edge 3 in Figure 1a indicates a def-use relation between the definition and a reference of the PHP variable $id.	**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).
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 $x.	**F4.** A reference r in language L_1 and a definition d in a different language L_2 have a *cross-language information-flow relation* if r generates r^*, and r^* forms the code that is used in the computation for the value of d. For example, in the PHP code echo "<input name='input1' value ='$x'>", the value of the PHP variable $x is assigned to the value of the HTML input 'input1'.

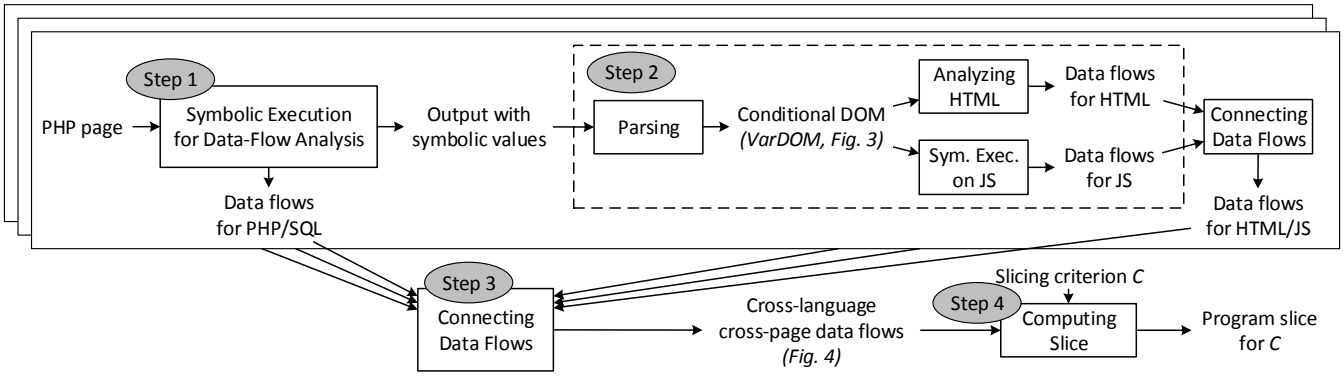

Figure 2: Approach overview

can always be easily expanded from a thin slice [46]; we discuss this expansion at the end of Section 5. Specifically, we define a (forward, thin) program slice with respect to a slicing criterion C (specified by the code location of a data entity) as *a set of definitions and references* of data entities in the web application that have *direct or indirect data-flow relations* from the value computed at C. For instance, the program slice for the (SQL) data entity sbulletinid on line 7 of Figure 1a includes all the data entities along the edges 1–10.

Types of data-flow relations: We propose a program-slicing technique for dynamic web applications that is based on the relations between the definitions and references of data entities, namely definition-use (*def-use*) relations and information-flow (*info-flow*) relations [7], which are traditionally used for analyzing programs written in a single language. In the context of dynamic web applications, we extend these relations also for entities that are written in different languages (see Table 1).

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 [36]. 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 1a 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 α // $_POST['edit'] == 1
2      <div>Database updated.</div>
3  #endif
4   ... <form name='announcements' ... >
5  #if β // mysql_fetch_array ($query)
6      <input ...  name='update[]' value='θ'> //θ represents $id
7  #endif
8  <input type='submit' value='Edit'></form>...
```

Symbolic execution explores different paths in a PHP program and computes/propagates the values of definitions and references of data entities. Conveniently, this process allows us to track the data flows within the server-side code. Since we need our symbolic execution engine anyway to approximate the output, we reuse and extend it with new mechanisms to record the data flows within PHP as well as SQL code, which is embedded in PHP strings and is also resolved by symbolic execution. In addition, an advantage of using symbolic execution is that we can eliminate some infeasible flows

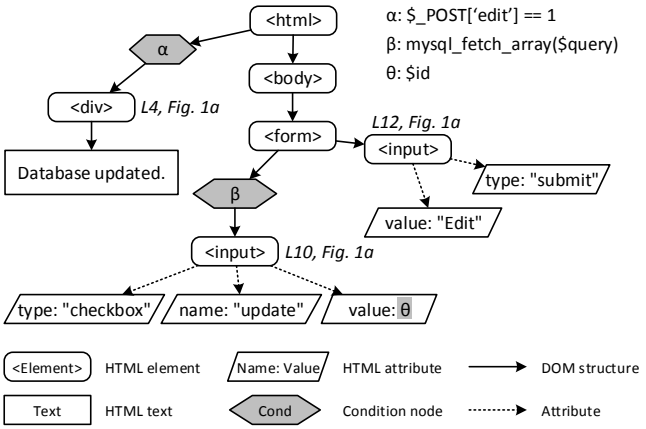

Figure 3: The VarDOM representation for the output of the program in Figure 1a

by checking the satisfiability of the path constraints under which the data entities appear.

```
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>';
```

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 [34]. The parsers were developed using the TypeChef variability-aware parser framework [25], which enables parsing programs with conditional 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 possible 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 1a. Unlike an HTML DOM, a VarDOM contains *condition nodes* to indicate that certain VarDOM elements 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 constraint β (mysql_fetch_array($query)), and its value attribute has a symbolic value θ ($id). 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.

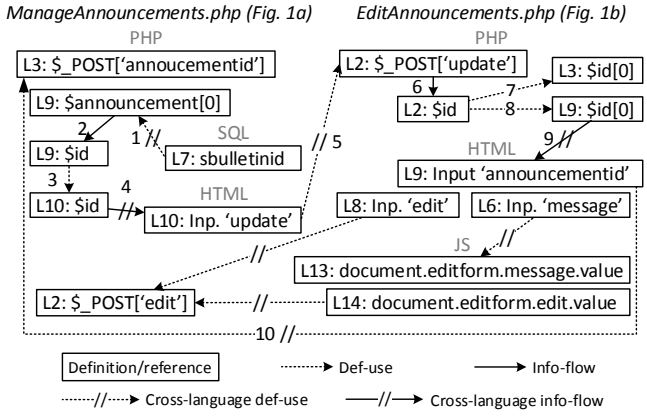

Figure 4: Data-flow graph (excerpt) for the example of Figure 1

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 variables 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/references 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 definition 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 [36]. 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., $s \in S$ 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 $(\mathcal{V}, \mathcal{D}, \pi)$ where the **value store** $\mathcal{V} : N \mapsto V$ is a (total) function mapping a variable/function name to its value (uninitialized variables have a $\perp$ value), the **path constraint** π encodes the branch decisions taken to reach the current state. In addition to $\mathcal{V}$ and π, which are typical for symbolic executors, to detect data flows, we track a **definition store** $\mathcal{D} : N \mapsto \mathcal{P}(L \times \Pi)$ that maps each variable name to its set of definitions together with a path constraint under which each definition appears.

Initialization:

$$\mathcal{V}(x) = \bot \quad \mathcal{D}(x) = \emptyset \quad \pi = \mathsf{TRUE}$$

1. Variable Access:

$$\frac{r = addEntity(\$n) \quad addRelation(d, r),\ \forall (d, \pi_d) \in \mathcal{D}(n), isSat(\pi_d \wedge \pi)}{\langle \$n, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle \mathcal{V}(n), \mathcal{V}, \mathcal{D}, \pi \rangle}$$

2. Assignment:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle \quad d = addEntity(\$n) \quad addRelation(r, d),\ \forall r \in vars(e) \quad \mathcal{D}'' = \mathcal{D}'[n \mapsto \{(d, \pi)\}]}{\langle \$n = e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}'[n \mapsto v], \mathcal{D}'', \pi \rangle}$$

3. If Statement:

$$\frac{\begin{array}{c}\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle \\ \pi' = whenEqual(v, \mathsf{TRUE}) \quad isSat(\pi \wedge \pi') \quad isSat(\pi \wedge \neg \pi') \\ \langle\!\langle s_1, \mathcal{V}', \mathcal{D}', \pi \wedge \pi' \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \mathcal{D}_1, \pi \wedge \pi' \rangle\!\rangle \\ \langle\!\langle s_2, \mathcal{V}', \mathcal{D}', \pi \wedge \neg \pi' \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \mathcal{D}_2, \pi \wedge \neg \pi' \rangle\!\rangle \\ \mathcal{V}_3(x) = ite(\pi', \mathcal{V}_1(x), \mathcal{V}_2(x)) \\ \mathcal{D}_3(x) = \{(d, \pi_d \wedge \pi') | (d, \pi_d) \in \mathcal{D}_1(x)\} \\ \bigcup \{(d, \pi_d \wedge \neg \pi') | (d, \pi_d) \in \mathcal{D}_2(x)\}\end{array}}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle ite(\pi', c_1, c_2), \mathcal{V}_3, \mathcal{D}_3, \pi \rangle\!\rangle}$$

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle \quad \pi' = whenEqual(v, \mathsf{TRUE}) \quad \neg isSat(\pi \wedge \neg \pi') \quad \langle\!\langle s_1, \mathcal{V}', \mathcal{D}', \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle\!\rangle}{\langle\!\langle \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle\!\rangle}$$

4. Function Declaration:

$$\frac{\lambda \text{ is a pointer to function } n(\$n_1, ..., \$n_m)\{s\}}{\langle\!\langle \mathsf{function}\ n(\$n_1, ..., \$n_m)\{s\}, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathsf{OK}, \mathcal{V}[n \mapsto \lambda], \mathcal{D}, \pi \rangle\!\rangle}$$

5. Function Invocation:

$$\frac{\begin{array}{c}\lambda = \mathcal{V}_0(n) \quad \lambda \text{ is a pointer to function } n(\$n_1, ..., \$n_m)\{s\} \\ \langle e_i, \mathcal{V}_{i-1}, \mathcal{D}_{i-1}, \pi \rangle \rightarrow \langle v_i, \mathcal{V}_i, \mathcal{D}_i, \pi \rangle,\ \forall i \in [1..m] \\ \mathcal{V}_f(x) = \begin{cases} v_i & \text{if } x = n_i \\ \bot & \text{otherwise} \end{cases} \\ d_i = addEntity(\$n_i),\ \forall i \in [1..m] \\ \mathcal{D}_f(x) = \begin{cases} (d_i, \pi) & \text{if } x = n_i \\ \emptyset & \text{otherwise} \end{cases} \\ addRelation(r, d_i),\ \forall r \in vars(e_i)\ \forall i \in [1..m] \\ \langle\!\langle s, \mathcal{V}_f, \mathcal{D}_f, \pi \rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}_{f'}, \mathcal{D}_{f'}, \pi \rangle\!\rangle \\ RET_{ref} = addEntity(n) \\ addRelation(RET_{def}, RET_{ref}), \\ \forall (RET_{def}, \pi_d) \in \mathcal{D}_{f'}(\text{`RET'})\end{array}}{\langle n(e_1, ..., e_m), \mathcal{V}_0, \mathcal{D}_0, \pi \rangle \rightarrow \langle \mathcal{V}_{f'}(\text{`RET'}), \mathcal{V}_m, \mathcal{D}_m, \pi \rangle}$$

6. Return Statement:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle \quad RET_{def} = addEntity(e) \quad addRelation(r, RET_{def}),\ \forall r \in vars(e) \quad \mathcal{D}'' = \mathcal{D}'[\text{`RET'} \mapsto \{(RET_{def}, \pi)\}]}{\langle\!\langle \mathsf{return}\ e, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathsf{RETURN}, \mathcal{V}'[\text{`RET'} \mapsto v], \mathcal{D}'', \pi \rangle\!\rangle}$$

7. Block of Statements:

$$\frac{\begin{array}{c}\langle\!\langle s_1, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c_1, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle\!\rangle \quad \pi' = whenEqual(c_1, \mathsf{OK}) \\ isSat(\pi \wedge \pi') \quad \langle\!\langle s_2, \mathcal{V}_1, \mathcal{D}_1, \pi \wedge \pi' \rangle\!\rangle \rightarrow \langle\!\langle c_2, \mathcal{V}_2, \mathcal{D}_2, \pi \wedge \pi' \rangle\!\rangle \\ \mathcal{V}_3(x) = ite(\pi', \mathcal{V}_2(x), \mathcal{V}_1(x)) \\ \mathcal{D}_3(x) = \{(d, \pi_d \wedge \pi') | (d, \pi_d) \in \mathcal{D}_2(x)\} \\ \bigcup \{(d, \pi_d \wedge \neg \pi') | (d, \pi_d) \in \mathcal{D}_1(x)\}\end{array}}{\langle\!\langle s_1 s_2, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle ite(\pi', c_2, c_1), \mathcal{V}_3, \mathcal{D}_3, \pi \rangle\!\rangle}$$

8. While Statement:

$$\frac{\langle\!\langle \mathsf{if}\ (e)\ \{s\ \mathsf{if}\ (e)\ s\}, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}', \mathcal{D}', \pi \rangle\!\rangle}{\langle\!\langle \mathsf{while}\ (e)\ s, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}', \mathcal{D}', \pi \rangle\!\rangle}$$

9. Include Expression:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle \quad \langle\!\langle s, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle\!\rangle \rightarrow \langle\!\langle c, \mathcal{V}_2, \mathcal{D}_2, \pi \rangle\!\rangle \quad s = \begin{cases} parseFile(v) & \text{if } v \text{ is a concrete value} \\ \text{empty statement} & \text{otherwise} \end{cases}}{\langle\!\langle \mathsf{include}\ e, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathcal{V}_2(\text{`RET'}), \mathcal{V}_2, \mathcal{D}_2, \pi \rangle\!\rangle}$$

10. Infix Expression:

$$\frac{\langle e_1, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v_1, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle \quad \langle e_2, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle \rightarrow \langle v_2, \mathcal{V}_2, \mathcal{D}_2, \pi \rangle \quad v = \begin{cases} concat(v_1, v_2) & \text{if } op \text{ is concatenation} \\ sym(e_1\ op\ e_2) & \text{otherwise} \end{cases}}{\langle e_1\ op\ e_2, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}_2, \mathcal{D}_2, \pi \rangle}$$

11. Echo Statement:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle \quad addOutput(v, \pi)}{\langle\!\langle \mathsf{echo}\ e, \mathcal{V}, \mathcal{D}, \pi \rangle\!\rangle \rightarrow \langle\!\langle \mathsf{OK}, \mathcal{V}', \mathcal{D}', \pi \rangle\!\rangle}$$

12. mysql_query:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle query, \mathcal{V}', \mathcal{D}', \pi \rangle}{\langle \mathsf{msql\_query}(e), \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle parseAndFindSqlDefs(query), \mathcal{V}', \mathcal{D}', \pi \rangle}$$

13. mysql_fetch_array:

$$\frac{\langle e, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle}{\langle \mathsf{msql\_fetch\_array}(e), \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi \rangle}$$

14. Array Access of SQL Data:

$$\frac{\langle e_1, \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle v_1, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle \quad \langle e_2, \mathcal{V}_1, \mathcal{D}_1, \pi \rangle \rightarrow \langle v_2, \mathcal{V}_2, \mathcal{D}_2, \pi \rangle \quad v_1 \text{ is a set of SQL definitions} \quad d \in v_1 \quad d \text{ has name/index } v_2 \quad r = addEntity(e_1[e_2]) \quad addRelation(d, r)}{\langle e_1[e_2], \mathcal{V}, \mathcal{D}, \pi \rangle \rightarrow \langle sym(e_1[e_2]), \mathcal{V}_2, \mathcal{D}_2, \pi \rangle}$$

Notation and auxiliary functions:
- $\mapsto$ denotes total functions.
- $g = f[x \mapsto y]$ denotes a function same as f except that $g(x) = y$.
- $sym(e)$ returns a fresh symbolic value mapped to an expression e.
- $ite(\pi, v_1, v_2)$ returns an *ite* value of v_1 and v_2 depending on π.
- $concat(v_1, v_2)$ returns a *concat* value of v_1 and v_2.
- $vars(e)$ returns a set of references appearing in an expression e (except for arguments of user-defined function calls).
- $isSat(\pi)$ returns TRUE if π is satisfiable and FALSE otherwise.
- $whenEqual(v, v')$ returns the constraint under which v equals v', e.g. $whenEqual(ite(\alpha, \mathsf{TRUE}, \mathsf{FALSE}), \mathsf{TRUE})$ returns α, $whenEqual(ite(\alpha > 1, \mathsf{TRUE}, \mathsf{FALSE}), \mathsf{TRUE})$ returns (fresh) β.
- $parseFile(v)$ parses a PHP file v and returns the parsed program.
- $parseAndFindSqlDefs(query)$ parses an SQL query and returns a set of SQL definitions (SQL table columns) in the query.
- $addOutput(v, \pi)$ records value v under constraint π in the output.
- $addEntity(e)$ creates and returns a new definition/reference from e.
- $addRelation(l_1, l_2)$ records a def-use/info-flow between l_1 and l_2.

Figure 5: Excerpt of PHP symbolic execution's evaluation rules (extensions to PhpSync [36] are highlighted in gray)

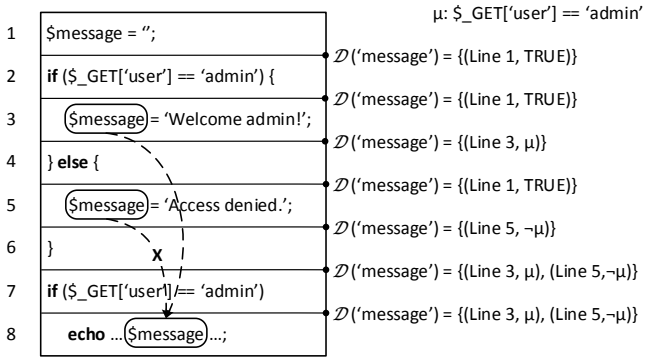

Figure 6: Detecting data flows at conditional statements

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 5 shows the key evaluation rules. For a statement s, a rule $\langle\langle s, \mathcal{V}, \mathcal{D}, \pi\rangle\rangle \rightarrow \langle\langle c, \mathcal{V}', \mathcal{D}', \pi'\rangle\rangle$ 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 c 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., 'RETURN') otherwise. For an expression e, a rule $\langle e, \mathcal{V}, \mathcal{D}, \pi\rangle \rightarrow \langle v, \mathcal{V}', \mathcal{D}', \pi'\rangle$ 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 [36] and highlight the parts that we extend to identify data-flow relations. *addEntity* and *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 *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 (**F1** and **F2** in Table 1). 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 π_r, we look up its definitions in the set $\mathcal{D}(n)$ and match π_r with the constraints of those definitions to retain only feasible relations. Specifically, a definition d with constraint π_d in $\mathcal{D}(n)$ has a feasible def-use relation with r if $\pi_d \wedge \pi_r$ 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/redefined 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.

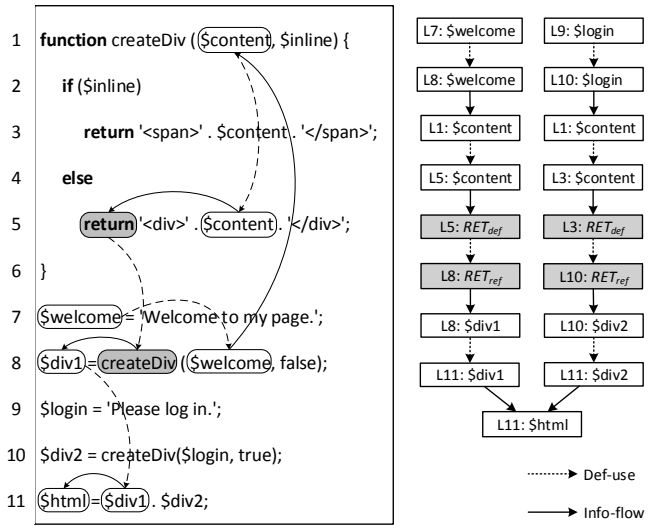

Figure 7: Interprocedural flows with highlighted RET nodes

Handling conditional statements (rule 3): If the path constraints of both branches of an if statement are satisfiable, we explore both branches. The function $whenEqual(v, \text{TRUE})$ 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 α == 'admin' where α is the symbolic value for $_GET['user']; thus, we evaluate the both conditions into the same (fresh) symbolic value μ (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 (and the value store) with the combined definitions (and values) from the two branches with their corresponding constraints. To represent that a variable may have multiple values depending on a path constraint, we use an $ite(\pi, v_1, v_2)$ value (short for *if-then-else*) to denote a selection between value v_1 if the path constraint π evaluates to TRUE and v_2 otherwise (we also use 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 μ 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.

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, 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

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* [47], 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 interprocedual 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 = $y + 1;
3      $y = $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 $\mathcal{V}$ and $\mathcal{D}$). 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.)

```
1  $x = new Foo(); $x->a = 1; $y = $x;
2  echo $y->a;
```

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 1a, sbulletinid 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 β; 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).

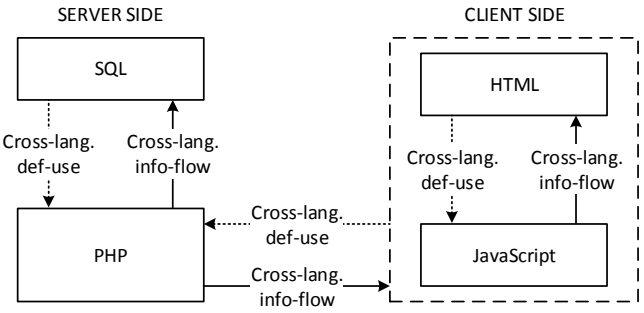

Figure 8: Data-flow relations across different languages

Table 2: Types of cross-language def-use relations

	Ref.	Definition	Reference example	Definition example
1	PHP	HTML input	**$_GET['input1']**	<input name='input1' **value**='0'...>
2	PHP	HTML URL	**$_GET['input1']**	<a href='index.php?**input1**=0'>
3	PHP	JS	**$_GET['input1']**	**document.form1.input1.value**='0'
4	PHP	SQL	**$row['column1']**	SELECT **column1** FROM table1
	JS	HTML by name:		
5	JS	- form	document.**form1**	<form name='**form1**'...>
6	JS	- input	doc...form1.**input1**	<input name='**input1**'...>
7	JS	- input value	doc...input1.**value**	<input name='input1' **value**='0'>
8	JS	HTML by ID	document. getEle...ById('**id1**')	

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.,).

(3) HTML definitions by URL parameters: These entities are detected in HTML query strings (e.g., the data entity lang in <a href = 'index.php?*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 8, we show all possible def-use and information-flow relations across languages. We detect those 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 $_POST/$_GET or $_REQUEST 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 [36] 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.

Table 4: Complexity of data-flow graph

System	PHP entities	Non-PHP entities						Data-flow edges					
		Total	SQL	HTML	JS	Embed.	N-Echo	Total	xLang	xFile	xFunc	xString	xPage
AB	10,591	266	8	220	38	46	10	13,406	416	2,538	3,043	474	356
			3%	83%	14%	17%	4%		3%	19%	23%	4%	3%
SM	4,935	2,402	404	729	1,269	2,402	452	6,945	2,426	1,565	164	2,603	1,292
			17%	30%	53%	100%	19%		35%	23%	2%	37%	19%
TC	15,291	2,145	214	1,717	214	2,145	214	16,490	655	3,493	46	937	332
			10%	80%	10%	100%	10%		4%	21%	0%	6%	2%
UPB	32,309	1,308	0*	1,160	148	1,191	447	36,186	4,983	6,986	3,121	5,470	4,886
			0%	89%	11%	91%	34%		14%	19%	9%	15%	14%
WC	3,805	497	48	377	72	86	48	3,934	887	1,463	1,056	992	829
			10%	76%	14%	17%	10%		23%	37%	27%	25%	21%
Total	**66,931**	**6,618**	**674**	**4,203**	**1,741**	**5,870**	**1,171**	**76,961**	**9,367**	**16,045**	**7,430**	**10,476**	**7,695**
			10%	**64%**	**26%**	**89%**	**18%**		**12%**	**21%**	**10%**	**14%**	**10%**

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*

**There are 0 SQL entities in UPB since this system stores data in local files instead of an SQL database.*

Table 3: Subject systems

Subject System		Size		Exec.		
Name	Version	Files	LOC	Entries	Stmts	Time
AddressBook (AB)	6.2.12	100	18,874	17	25,713	10.0s
SchoolMate (SM)	1.5.4	63	8,183	1	2,942	5.2s
TimeClock (TC)	1.04	69	23,403	32	26,388	13.3s
UPB	2.2.7	395	104,640	51	77,959	37.6s
WebChess (WC)	1.0.0	39	8,589	9	6,874	4.6s

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 [3], 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

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 that slices are very difficult to manually identify or that developers could significantly benefit from a tool in complicated (favorable) cases. However, slicing in general has been shown to be useful in a number of applications [47, 51]. The more interesting question is 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 5: Complexity of slices

System	Slices	Size	Len	xLang	xFile	xFunc	xString	xPage
AB	6,827	6	5	287	2,330	3,202	344	243
SM	4,185	5	4	1,518	1,519	917	1,735	890
TC	9,145	6	4	1,193	2,007	643	1,378	224
UPB	17,906	7	5	795	7,904	8,386	1,236	681
WC	2,607	4	3	312	1,557	1,517	408	265
Tot/Avg	**40,670**	**5.6**	**4.2**	**4,105**	**15,317**	**14,665**	**5,101**	**2,303**
				10%	**38%**	**36%**	**13%**	**6%**

Size, Len: Median size and length of a slice; xLang, xFile, xFunc, xString, xPage: Slices that cross languages, files, functions, strings, and page entries

9.1 Experiment Setup

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.

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 inlined 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.

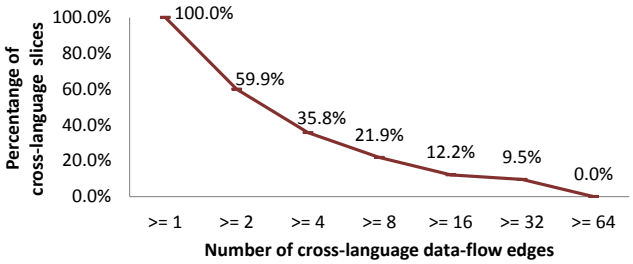

Figure 9: Cross-language data flows in a cross-language slice

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, 8, 18, 19, 28, 20, 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 extrusion, 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 [38], call-mark slicing [37], proposition-based slicing [22], stop-list slicing [15], 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 [27, 9, 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.

There exist much research on exploring flows among Web pages for testing [6, 31, 44] or code comprehension [13]. However, they do not build *cross-language*, *cross-stage* slices as in WebSlice.

There exist string analysis approaches for web programs and software security [32, 26, 49, 50, 52, 4]. They can be used to extract embedded code in our analysis. Maule *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

[1] Cross-language program slicing for dynamic web applications. `http://home.engineering.iastate.edu/~hungnv/Research/WebSlice`.

[2] JavaBDD website. `http://javabdd.sourceforge.net/`.

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

[4] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering*, pages 947–957. IEEE Press, 2012.

[5] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance*, pages 292–301. IEEE Computer Society, 1993.

[6] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of Javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 571–580. ACM, 2011.

[7] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, Jan. 1985.

[8] D. Binkley and K. Gallagher. Program slicing. *Journal of Advanced Computing*, 43:1–50, 1996.

[9] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2014.

[10] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Journal of Advanced Computing*, 62:105–178, 2004.

[11] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Inf. Soft. Technology*, 40(11-12):595–608, 1998.

[12] A. de Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th International Workshop on Program Comprehension*, pages 9–18. IEEE Computer Society, 1996.

[13] G. A. Di Lucca and M. Di Penta. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, pages 87–94. IEEE Computer Society, 2005.

[14] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379–392. ACM, 1995.

[15] K. Gallagher, D. Binkley, and M. Harman. Stop-list slicing. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 11–20. IEEE Computer Society, 2006.

[16] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 323–334. ACM, 2012.

[17] M. Harman and S. Danicic. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 70–79. IEEE Computer Society, 1997.

[18] M. Harman, S. Danicic, Y. Sivagurunathan, and D. Simpson. The next 700 slicing criteria. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*, 1996.

[19] M. Harman and K. Gallagher. Program slicing. *Inform. Softw. Technol.*, 40:577–582, 1998.

[20] M. Harman and R. Hierons. An overview of program slicing. *Softw. Focus*, 3:85–92, 2001.

[21] M. Harman, R. Hierons, C. Fox, S. Danicic, and J. Howroyd. Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 138–147. IEEE Computer Society, 2001.

[22] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher Order Symbolic Computation*, 13(4):315–353, 2000.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 35–46. ACM, 1988.

[24] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–47. ACM, 2005.

[25] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 805–824. ACM, 2011.

[26] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, pages 199–209. IEEE Computer Society, 2009.

[27] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, Oct. 1988.

[28] A. D. Lucia. Program slicing: Methods and applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society, 2001.

[29] J. Maras, J. Carlson, and I. Crnkovic. Client-side web application slicing. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 504–507. IEEE Press, 2011.

[30] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *Proceedings of the 30th International Conference on Software Engineering*, pages 451–460. ACM, 2008.

[31] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, pages 210–220. IEEE Computer Society, 2009.

[32] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 432–441, 2005.

[33] M. N. Ngo and H. B. K. Tan. Applying static analysis for automated extraction of database interactions in web applications. *Information and Software Technology*, 50(3):160–175, 2008.

[34] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 518–529. ACM, 2014.

[35] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. Nguyen. Dangling references in multi-configuration

and dynamic PHP-based web applications. In *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 399–409. IEEE Press, 2013.

[36] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 13–22. IEEE Computer Society, 2011.

[37] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue. Call-mark slicing: An efficient and economical way of reducing slice. In *Proceedings of the 21st International Conference on Software Engineering*, pages 422–431. ACM, 1999.

[38] A. Orso, S. Sinha, and M. Harrold. Incremental slicing based on data-dependence types. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 158–167. IEEE Computer Society, 2001.

[39] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM, 1984.

[40] F. Ricca and P. Tonella. Web application slicing. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 148–157. IEEE Computer Society, 2001.

[41] F. Ricca and P. Tonella. Construction of the system dependence graph for web application slicing. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 123–132. IEEE Press, 2002.

[42] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, pages 277–287. IEEE Press, 2012.

[43] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, 2010.

[44] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *Proceedings of the 2013 9th Joint ESEC/FSE Meeting on Foundations of Software Engineering*, pages 422–432. ACM, 2013.

[45] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.

[46] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122. ACM, 2007.

[47] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, 1994.

[48] P. Tonella and F. Ricca. Web application slicing in presence of dynamic code generation. *Journal of Automated Software Engineering*, 12(2):259–288, 2005.

[49] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, pages 171–180. ACM, 2008.

[50] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*. USENIX Association, 2006.

[51] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[52] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 251–260. IEEE Press, 2011.