Dangling References in Multi-configuration and Dynamic PHP-Based Web Applications

Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University, USA

Abstract—PHP is a dynamic language popularly used in Web development for writing server-side code to dynamically create multiple versions of client-side pages at run time for different configurations. A PHP program contains code to be executed or produced for multiple configurations/versions. That dynamism and multi-configuration nature leads to dangling references. Specifically, in the execution for a configuration, a reference to a variable or a call to a function is dangling if its corresponding declaration cannot be found. We conducted an exploratory study to confirm the existence of such dangling reference errors including dangling cross-language and embedded references in the client-side HTML/JavaScript code and in data-accessing SQL code that are embedded in scattered PHP code. Dangling references have caused run-time fatal failures and security vulnerabilities.

We developed DRC, a static analysis method to detect such dangling references. DRC uses symbolic execution to collect PHP declarations/references and to approximate all versions of the generated output, and then extracts embedded declarations/references. It associates each detected declaration/reference with a conditional constraint that represents the execution paths (i.e., configurations/versions) containing that declaration/reference. It then validates references against declarations via a novel dangling reference detection algorithm. Our empirical evaluation shows that DRC detects dangling references with high accuracy. It revealed 83 yet undiscovered defects caused by dangling references.

Index Terms—Dangling References; Web Code Analysis

I. INTRODUCTION

To accommodate multiple usage scenarios and configurations in Web applications, developers often use a dynamic scripting language to write server-side code to dynamically produce multiple versions of client-side pages at run time. A popular Web language is PHP: 54% of all Web-based projects in Sourceforge [27] use PHP. A PHP server program contains the code to be executed or created for multiple configurations, usage scenarios, or versions. For example, it contains PHP code to process different accessing scenarios for different types of users (e.g., regular or admin users) or Web browsers (Internet Explorer, Mozilla, or Chrome). A PHP program also contains client-side code in HTML or JavaScript (JS) embedded in PHP strings for different versions of client pages such as for different languages. Thus, parts of a PHP server-side program will be executed for the same configuration, usage scenario, or version; while other parts might not. Let us call this the multi-configuration nature of PHP code.

The multi-configuration nature and dynamism in PHP often cause Web programmers to make errors in dealing with PHP code corresponding to different configurations, scenarios, or versions. One kind of errors that one could encounter happens when the usage/reference to a variable corresponding to a configuration/scenario/version does not have its respective declaration in the current execution for that configuration. Let us call them dangling or undefined reference errors (dangling references for short). The same occurs with PHP function calls where an invoked function is undefined in the current run.

To investigate this type of dangling reference errors, we conducted an empirical study in which we manually examined 477 selected revisions of seven PHP-based Web applications. We found 103 revisions containing the fixes for a total of 163 instances of such dangling references. Interestingly, we found the defects caused by the dangling references occurring in embedded code written in HTML, JS, and SQL. In a PHP server program, client-side code in HTML/JS, and data-accessing code in SQL are often embedded within PHP string literals and dynamically generated at run time. Those references can also be dangling, and we call them dangling embedded references.

For example, a PHP function $F$ is used to generate an HTML text input as follows: echo "<input name="userid">". That HTML $userid$ element is referred to by a JS reference via a DOM-based syntax as in echo "if (document.loginform.userid.value!=...)". In this case, both declaration and reference are embedded in PHP strings. At run time, if the PHP function $F$ is not invoked and the JS reference is generated, then that JS reference will be dangling because the $userid$ element does not exist. Importantly, there are cross-language references, e.g., from JS to HTML, PHP to HTML, PHP to SQL, etc.

In brief, such dangling references occur at run time in the following: 1) references to PHP variables or calls to PHP functions that have not been properly defined (i.e. undefined PHP variables/functions), 2) embedded references to unavailable database fields in the result sets of SQL queries that were not retrieved, 3) calls in embedded code to JS functions that were not declared, and 4) references to generated HTML elements that have not been initialized in the current execution.

We also found that those dangling references have caused serious run-time failures ranging from warnings, incorrect behaviors, to security vulnerabilities such as denial of services, input validation bypass, and fatal crashes. Thus, it is desired to detect them early. In a traditional, statically typed language, a reference is dangling if it does not refer to any program entity in a static scope of a program. It can be detected at compile time by a compiler. However, PHP’s dynamism makes it challenging to statically detect such dangling references, i.e., statically determining if there exists an execution such that
a reference does not have its respective declaration, since a reference might be dangling only in some configuration(s) or version(s), and might not be so in others. It is also challenging to statically decide from PHP code which versions a specific embedded declaration/reference generates and if a corresponding reference has a respective declaration, since a PHP interpreter embedded declaration/reference generates it. It is also challenging since a generated version(s), and might not be so in others. In this work, we developed Dangling Reference Checker, DRC, a static analysis method to detect such dangling references in dynamic PHP code. DRC first parses a PHP program and recognizes the declarations/references of PHP variables and functions. Second, it performs symbolic execution [16] on PHP code, to create a D-model, a tree-based representation approximating all possible versions of the generated output, and then analyzes the texts in the D-model with a specialized parser to extract embedded references/declarations in HTML/JS/SQL. It associates each reference declaración with a constraint that represents the symbolic execution path of the configuration/scenario or the version of client code having that declaration/reference. Finally, our dangling reference detection algorithm searches for each reference a corresponding declaration with a matched signature (i.e., belonging to the same entity) and with a consistent constraint (i.e., belonging to the same execution path for a configuration or a client version). If no such declaration exists, the reference is seen as dangling.

Our empirical evaluation on several Web applications shows that DRC achieves an average of 89% recall (up to 100%) and 71% precision (up to 80%) in detecting PHP and embedded dangling references. Moreover, it revealed 83 not-yet-discovered faults caused by such dangling references. Our key contributions in this paper include:

1) An exploratory empirical study and findings on a type of errors caused by dangling PHP and embedded references.
2) The new concept of embedded references in Web code.
3) A static analysis tool to detect dangling PHP and embedded references in multi-configuration, dynamic PHP code.
4) An empirical evaluation on several large, real-world Web applications to show DRC’s correctness and efficiency.

II. AN EXPLORATORY STUDY

Aiming to understand the problem of dangling references in multi-configuration, dynamic PHP code, we conducted an empirical study on seven PHP-based, dynamic Web applications. Our key research questions in this study include:

1. Do dangling references in PHP and embedded code exist in dynamic Web applications?
2. What are the causes of such dangling references?
3. What are the types of failures that they cause?

A. Data Collection and Results

We selected as our subject systems seven active PHP-based Web projects in SourceForge (Table I) with high popularity rates and long histories. Column Start shows their starting dates. To identify the candidate revisions (column Cand.Rev) that potentially contain the fixes for dangling bugs, we wrote a program to analyze the commit logs of all revisions (up to 07/11/12), and selected the ones containing the keywords such as “dangling”, “dangled”, “undefined”, and “undeclared”. To remove false positives, we then manually investigated all 477 revisions and collected those that actually contained a fix to at least one dangling reference error (column DanglingRev).

The last 4 columns show the numbers of cases that occurred in PHP code (PHP) and embedded code (Embed). For each type, columns Ent and Ref display the number of entities (variables/functions) having dangling references and the number of the dangling references, respectively. (One entity might have multiple references to it.) As seen in Table I, 103 revisions were reported as the fixes for at least one dangling bug. There were 163 dangling references, with 146 and 17 being PHP and embedded ones. Let us explain some examples in the projects to illustrate different types of dangling references.

B. Examples of Dangling PHP and Embedded References

1) PHP Dangling Reference $group_value in ImpressCMS Project at Revision 4700 (Figure 1a): To create permission controls for different user groups, the function createPermissionControls selects
a group based on the value of the variable $groups_value (line 270). The variable $groups_value is defined earlier depending on several conditions. If the target object is new (COND1 = true, line 263), the value will be retrieved from a configuration array (lines 264-266), after the existence of the configuration item is checked at line 264 (COND2). Otherwise, if the group’s value is already stored in the target object (COND1 = false), $groups_value will be assigned with this stored value (line 268).

As illustrated in Figure 1b, there are three possible execution paths (corresponding to three scenarios/configurations) that lead to the use of the variable $groups_value at line 270. In one of those paths (COND1 = T and COND2 = F), the variable is not defined, and the reference to $groups_value at line 270 becomes dangling. The commit log below the code in Figure 1a confirms this error. In the next revision, the developers fixed the error by adding a default value assignment for $groups_value after line 262 so that it is defined in all possible execution paths. (In PHP, a variable is defined when it is first assigned with a value.) It is challenging to statically detect this type of errors since in general, there might exist many paths with multiple nested if statements, and the declarations/assignments of variables and references could be far apart in multiple functions. Note that the failure occurred only in the configuration when COND1 = T and COND2 = F, but did not in others.

2) Embedded Dangling References in PHP-Fusion at Revision 2600 (Figure 2): Besides dangling PHP references, we found a new type of dangling references in embedded code. Figure 2 shows a code snippet for typical operations involved in a database query that selects a set of data rows from one or more database tables. First, the SQL query (lines 198-206) is sent to the database server and the query’s result will be stored in the PHP variable $result (line 197). Then, each row in the result data set will be retrieved for processing (line 212). For a data row $data, the data stored in different columns of the row (e.g., $data['user_post'] on line 262 and $data['user Joined'] on line 263) will be displayed. Note that the columns available for selection are specified in the SELECT part of the SQL query (lines 198-200). However, on line 264, there is an access to the SQL column user_lastvisit, which was not selected in the SQL query; thus, the reference to user_lastvisit is dangling. We call it a dangling embedded reference since the reference to a column in the query’s result is embedded within a PHP string ‘user_lastvisit’. This error was fixed by adding user_lastvisit to the list of selected columns (line 200). Generally, it is not trivial to statically detect dangling embedded references since the references are embedded in the strings resulted from several computation operations on scattered PHP strings or variables.

3) Cross-Language, Embedded Dangling Reference in PHP-Fusion at Revision 1650 (Figure 3): The code on lines 106-111 of Figure 3 will generate different HTML input entities named news_thumb_w and new_thumb_h. These input fields allow users to specify the width and height of a photo and its thumbnail. As a user submits this information, the code on lines 46-54 will access these values using the PHP special variable $POST and update the database. However, at revision 1650, due to a copy-and-paste error, the input names on lines 110 and 111 were accidentally the same as those on lines 106 and 107. Thus, the input fields for the height and width of a photo were missing. Since the HTML inputs news_photo_w and news_photo_h are not defined, the references to them on lines 51-54 are dangling. Such references are embedded in PHP strings (‘news_photo_h’ and ‘news_photo_w’) and refer to entities in a different language (i.e., HTML inputs). As seen, the developers fixed it by changing the duplicated names news_thumb_w and new_thumb_h to news_photo_w and news_photo_h.

C. Causes and Effects

To learn more about the causes of those dangling references, we examined the commit logs and changed code in those cases. The prominent reasons include:

1. Developers manually renamed the entities and missed an instance. For example, in the file start_left.php at revision 1875 of Beehive Forum, the variable $thread['id'] was renamed to $thread['ID']. A reference at line 156 was not changed, thus, became dangling. It was fixed at revision 1876. This suggests a need for better tool support for Web code refactoring.

2. Developers refactored code. After the code was restructured and a portion was moved, a variable assignment/declaration is not executed while its reference still is (i.e., making the declaration and reference belong to two different scenarios). Thus, that reference becomes dangling.
3. Errors due to copy-and-paste: see the example in Figure 3. The input names at lines 110-111 were not modified consistently, leading to dangling references at lines 51-54. This calls for a tool to help make consistent changes in cloned code.

4. The execution condition of a reference is not consistent with that of its declaration: a declaration and its reference are for different configurations (Figure 1).

5. Developers used a wrong or mistyped entity.

6. The ‘include’ statement of a file containing a declaration was misplaced such that it was not executed, making its references dangling. Similar to case 2 except that the declaration of a function is in a file and included at a wrong location, making it belong to a different scenario than that of the function calls.

To learn about the system failures caused by those dangling references, we examined the logs and the corresponding code. We found that they have caused run-time failures including:

1. Fatal errors and crashes: There were a total of 35 reported cases in which such dangling references (including undefined functions) have resulted in fatal errors.

2. Security vulnerabilities: Developers have recorded in the commit logs that such errors caused failures related to security such as denial of service (e.g. “...preventing the users accessing the forum” in Beehive Forum), and input validation bypass (e.g. the login scheme in PHPWiki).

3. Incorrect and unexpected behaviors: In general, the semantics of the Web program is broken as a reference or a function call cannot be used properly.

Conclusions. Our results confirmed that PHP and embedded dangling references (including undefined functions) exist in PHP Web applications due to several causes, and are a serious defect leading to run-time failures. To address that, we introduce DRC, a method to statically detect such dangling references. Let us first explain important concepts in DRC.

III. IMPORTANT CONCEPTS

DRC focuses on specific types of server-side elements and client-side elements. The former ones include PHP variables and functions, SQL fields and tables. The latter ones include HTML elements (forms, inputs, URLs, etc.) and JS variables and functions. An entity is an element with a name or an ID. Referencing is done via the elements’ names or IDs.

A declaration of an entity is the code where the entity is created/assigned. A reference to an entity is the code where the entity is referred to. A configuration/scenario is defined by a set of conditions on the program input. A configuration/scenario has its corresponding execution, which must satisfy its conditions. It is at run time and it differs from conditional compilation where a configuration has corresponding static code produced after pre-processing. A declaration/reference to a client-side entity (HTML/JS) or to an SQL entity might be embedded within PHP code. It appears in a certain version of the output of the PHP program that is produced via a particular execution of the program. The corresponding location in PHP code responsible for generating a client-side declaration/reference or SQL element is called an embedded declaration/reference. For example, in Figure 3, line 106 creates an HTML text input element named news_thumb_w. Line 47 contains a reference to it in $POST['news_thumb_w'].

When an element is referred to without proper declaration, it is called a dangling reference. An embedded reference $R_e$ (Table II) is said to refer to an embedded declaration $D_e$ if and only if at a particular version of the output of the PHP program, the reference $R$ generated from $R_e$ refers to the declaration $D$ generated from $D_e$. Similarly, a non-embedded reference $R$ is said to refer to an embedded declaration $D_e$ if $R$ refers to the declaration $D$ generated from $D_e$.

A (path) constraint for a declaration/reference is a set of conditions on the program state which needs to be satisfied for that entity to be exercised in an execution. Besides the constraint for an execution corresponding to a configuration/scenario, there is also a constraint for an execution to produce a version of client-side code. If a declaration/variable is used in a configuration or used to create a version of client code, its constraint is the constraint for that configuration/version. Importantly, we will match the constraints of declarations/references to detect dangling references (Section VII).

IV. PHP ENTITY DETECTION AND CONSTRAINTS

Let us present how DRC detects PHP entities (declarations/references). PHP code is symbolically executed via PhpSync [16] and entities are identified as follows.

Declarations of PHP entities. According to PHP language manual [28], a PHP variable is considered as declared when it is assigned with a value in a program execution such as in an assignment expression, a parameter of a function call, or a running variable of a for statement. For a PHP function, its declaration is the code location where the function is defined. References of PHP entities. When a PHP variable appears in non-declaring locations (e.g., the variable is not assigned with any value, but its value is accessed for an operation), DRC considers the variable’s location in the code as the reference to the corresponding variable. Similarly, an invocation of a PHP function is considered as a reference to that function.

Constraints of PHP entities. The constraint for a declaration/reference represents the condition which an execution
needs to satisfy for that declaration/reference to be exercised. That execution is for a specific configuration/scenario in the application. DRC uses path constraints in a symbolic execution to approximate such constraints. During its symbolic execution phase, it explores different execution paths of the PHP program. When a declaration/reference of a PHP entity is found, DRC associates it with the current path constraints. Figure 1b illustrates the constraints of the declarations and references of PHP variable $groups_value in Figure 1a. These constraints and the PHP entities are stored in an Entity Table (Table III). For example, the constraint for the declaration of the PHP variable $groups_value at line 265 of the file icmsform.php in Figure 1a is $COND1 AND $COND2, where $COND1 is $this->targetObject->isNew() and $COND2 is isset($thissModuleConfig['def_perm']).

V. EMBEDDED HTML/JS ENTITY DETECTION

DRC also detects HTML/JS entities embedded in PHP code. DRC performs symbolic execution on PHP code to approximate all possible output versions of the PHP code by a single tree and then detects entities on that tree.

A. D-Model Representation of Program Output

During its symbolic execution with PhpSync [16] to detect PHP entities (Section IV), DRC creates a tree-based representation, called D-model, which approximates all possible textual outputs of a PHP program. Details on PhpSync can be found in [16]. Let us illustrate the idea via an example. Figure 4 shows a simple login page that can be displayed in either English or German. Figure 5 represents the output of that PHP page by a D-model rooted at a Concat node. That Concat node represents a concatenation of multiple sub-strings, three of which are concrete string values (represented by string literal nodes), and one is a string value selected from two alternatives (represented by the two sub-trees under the Select node). These two alternatives correspond to the cases where the language option of the page equals ‘de’ (German) or not. Each subtree represents the concatenation of other sub-strings.

During the symbolic execution, DRC maintains the mapping between the D-model’s leaf nodes and their PHP locations. For example, the literal node userid under the first Concat node of the Select node is mapped to the characters on line 13 of Figure 4. The mapping is used in the entity detection step.

B. HTML Parsing on D-Model

Given a D-model that represents all possible versions of the client page, DRC analyzes it to identify HTML/JS elements on the multi-version client page. We design our HTML parser for D-model with two key differences from a regular parser:

1. A regular HTML parser works on a single version of the client page, whereas DRC’s HTML parser takes as input a D-model tree and parses fragmented HTML code contained in the leaf nodes of the tree.
Figure 6 shows DRC’s algorithm to parse a D-model. It traverses the D-model tree in the left-to-right order and handles different types of D-model nodes as follows.

1. **For a Concat node**, DRC visits the child nodes of the Concat node and parses them sequentially (lines 3-4). The parser uses the ending lexical state after parsing a child node as the starting lexical state when parsing the next child node.

2. **For a Repeat node** (which represents a string value that can be repeated multiple times), its child node is a D-model representing the repeated output value. For example, the output of the following code snippet is modeled by a Repeat node whose child node is a Literal node “<br/>”. Aiming to detect embedded entities, DRC only needs to analyze the child node of a Repeat node once (line 7).

3. **For a Literal node**, DRC first parses the HTML code fragment contained in the Literal node into HTML tokens (line 10). DRC has the same parsing strategy as a regular HTML parser for such HTML fragments except for the parsing of HTML attributes and their values where HTML entities’ names and IDs can be found. Figure 7 shows the state transitions of an automaton to recognize HTML attributes and values. The label of a transition edge contains the input characters (or regular expressions) in the first line, and the recognized HTML token in the second line (if any). For instance, given an HTML fragment `<input name='username' ...>` after reading `<input`, the parser transitions from the state TAG/TEXT to the state ATTR_NAME, recognizing an OpeningTag token and expecting the next characters as an AttrName token.

In the second step (line 11, Figure 6), DRC parses those HTML tokens to identify HTML elements and their attributes’ names and values. For example, after tokenizing the string `<input name='username' ...>`, it recognizes an HTML input element with an attribute `name` whose value is ‘username’. If an attribute value (representing the name of an HTML entity) is split up and belong to two leaf nodes, DRC maintains the same state ATTR_VAL to parse the next leaf node. It then detects an attribute value as an entity’s name by combining constituent substrings in two leaf nodes after encountering the ending single or double quote in that next leaf node.

4. **For Select nodes**, DRC does not derive all combinations of the two branches of all Select nodes, which would face a combinatorial explosion. Instead, it considers sequential Select nodes as independent and parses each of them individually. For a Select node, DRC records the lexical state before parsing the first branch (lines 14-15, Figure 6), and after that it backtracks to that state to continue parsing the other branch (lines 16-17). After visiting the two branches of a Select node, it maintains only one lexical state, which is shared between them, to continue parsing the next D-model node. The two lexical states after parsing two branches are usually the same since the HTML code fragments in the two branches need to fit with the same HTML code that follows right after them. Even when the two branches contained HTML elements of different types (e.g., one branch has an HTML form, and the other is an HTML table), the two branches would still end with the same state TAG. Thus, the number of traversed branches is only linear to the number of Select nodes in the D-model.

Let us illustrate such parsing process via an example. The D-model in Figure 8a is a simplified version of the D-model of Login.php (Figure 5). DRC begins by visiting the first literal node under the Concat node and parses the string `<form name='loginform' >`. At this step, it recognizes an HTML `<form>` tag element with the name loginform. When parsing the two branches of the Select node, it finds two `<input>` entities with the same name ‘userid’. These two HTML input entities belong to the HTML form entity ‘loginform’ but correspond to two different versions of the client page. Using the mapping information stored in the D-model, the two names ‘userid’ of the entities are then mapped to their corresponding locations in the server code (lines 13 and 16 of Figure 4a).

C. Extracting Client-Side Entities

During the parsing of the D-model, DRC collects client-side declarations/references into the Entity Table as follows:

1. **Declarations of HTML entities.** HTML entity declarations are identified by the ‘name’ or ‘id’ attributes of HTML elements, or can be found in HTML query strings. For
instance, the HTML code `<input name='userid'.../>` specifies a declaration of an HTML input named ‘userid’.

2. Declarations and references of JS entities, and JS references to HTML entities. While parsing HTML code, DRC extracts any JS code embedded in it. These locations include HTML `<script>` tags and HTML event handlers (e.g., `onload`, `onclick`). We customized Eclipse’s JS parser to detect from JS code any declarations/references of JS entities (i.e., function declarations/calls), or JS references to HTML entities (via HTML DOM syntax, e.g., `document.getElementById`).

Scopes of client-side entities. The client-side entities except HTML input entities have a global scope since they can be directly accessed from anywhere in the client- or server-side code. For an HTML input, its scope is defined by the HTML form that it belongs to. If two HTML input declarations with the same name are in different HTML forms/scopes, they will define two different entities. Similarly, the JS code `document.form1.userid` and `document.form2.userid` specifies two references to different HTML input entities with the same name `userid` (one belongs to `form1` and the other belongs to `form2`).

Constraints of client-side entities. Similar to the path constraints of PHP entities, the constraints of HTML/JS entities are the conditions in the symbolic execution that generate the entities. For example, the constraint for the `userid` HTML input at line 13 of Figure 4 is `@lang="de"`. That is, that HTML input will be generated in a version of client-side code when the language option is not German. Therefore, for a client-side declaration/reference, DRC constructs its constraint by combining (via the AND Boolean operator) all the conditions at the Select nodes in the D-model’s path that leads to the literal node containing that declaration/reference.

PHP references to HTML entities. PHP references to HTML entities can be found in pre-defined PHP variables such as `$_REQUEST`, `$_POST`, or `$_GET`. For example, `$_GET['userid']` is a reference to the HTML entity ‘userid’. For PHP references to HTML, their scope is global and no constraint is maintained.

VI. EMBEDDED SQL ENTITY DETECTION

A dynamic Web application often contains SQL code embedded within PHP strings and stored in variables. A PHP program does not produce SQL code to be displayed/run in a browser. Instead, it forms an SQL query via one or multiple PHP expressions and string literals and passes it to a database engine to be executed. The procedure to retrieve and display data from a database via SQL queries is as follows:

1. Sending an SQL query to a database: To do so, a PHP function, e.g. `mysql_query`, is executed. To get data from a database table, the SQL command `SELECT` is used (e.g. `SELECT column_name(s) FROM table_name`). The result is a record set containing the table’s data and is stored in a PHP variable.

2. Accessing data records/rows: A PHP function, e.g. `mysql_fetch_array`, is called on the record set to access its records/rows. Each call returns a single row, which is often stored in an array. Then be accessed by using the column name as a key of the corresponding PHP array containing the data row.

Figure 9 illustrates those steps. The PHP code with SQL queries is aimed to display a product name and a vendor name given their IDs. In step 1, after the SQL queries are sent to look up the two database tables products and vendors, the returned results are stored in the variable `$result` (lines 1 and 3, respectively). In step 2, the function `mysql_fetch_array` is called to get a row in the returned record set, and the results are stored in the associative arrays `$product` and `$vendor` (lines 2 and 4). Then, the names of the product and the vendor are printed out using the keys name of the corresponding arrays (line 5).

In the above process, the first step can be viewed as the declaration of a record set returned from a SQL query, whereas the second step specifies the references to the data column in the record set. The definitions of SQL entities, declarations and references are based on such a record set as follows.

Definition 1 (SQL entity). An SQL entity is a data column in the record set returned from an SQL SELECT query.

Definition 2 (SQL entity declaration). The declaration of an SQL entity is a column name or an alias column name in the record set corresponding to the SQL entity.

Definition 3 (SQL entity reference). A reference to an SQL entity is a key in an associative array which specifies a column name of the record set that is stored in the array.

For example, the identifiers name at D1 and D4 in Figure 9 are the declarations of two SQL entities. The array keys name at D7 and D8 are references to those SQL entities, respectively.

Scopes SQL Entities. It is possible that two SQL references having the same name could refer to different SQL entities. For instance, in Figure 9, the two references at D7 and D8 use the same identifier name but refer to two different entities at D1 and D4, since they are involved with two different record sets returned at D2 and D5, respectively. To distinguish such SQL references, DRC takes into account their program scopes.

Definition 4 (SQL entity scope). The scope of an SQL entity declaration, or reference is the life time of the corresponding record set of the SQL entity during the execution.

By this definition, a newly-created record set defines new scopes for one or multiple SQL entities. The SQL declaration and references of an SQL entity have the same scope defined.
by the record set of the SQL entity. In that scope, the SQL declaration has effect and SQL references are valid to refer to the entity. For example, the declaration at D1 and references at D2, D3, and D7 belong to the same scope defined for the data column name of the record set returned by the query at line 1. DRC will determine that they belong to the same scope via their data dependencies. Scopes corresponding to different record sets are different. For example, the SQL declaration/references at D4, D5, D6, D8 belong to another scope for the query at line 3. Since the two references at D7 and D8 are in different scopes, they refer to different SQL entities despite using the same identifier name.

**SQL Entity Detection Algorithm.** To extract SQL entities embedded in PHP code, DRC looks for PHP expressions and variables that access the data retrieved from the database and analyzes their data dependencies. Figure 10 shows our embedded SQL declaration/reference detection algorithm. DRC symbolically executes a PHP program and handles specially for three types of expressions relating to accessing database data. First, at a PHP `mysql_query` statement (line 5), it computes the D-model for its SQL query. Then, it uses a simple SQL parser similar to its HTML parser to identify SQL entity declarations in the D-model (lines 6-7). These declarations are the column names found in the SELECT part of an SQL SELECT query (e.g., `SELECT FROM` products) or alias column names found in the AS part (e.g., `SELECT product AS name FROM`). Newly-found declarations are associated with the scope defined by the returned record set (lines 8-11).

Second, as a row in the record set is accessed (via the functions `mysql_fetch_array`, `mysql_fetch_row`, or `mysql_fetch_assoc`), DRC records data dependency by assigning the scope of the returned array with that of the record set (lines 12-13). Third, for an array access, DRC first checks the array’s scope to see whether it is a record set returned from a database query (line 15). In that case, DRC identifies the key in the array access as a new SQL reference and associates it with the scope of the array (lines 16-17). The scopes for those SQL entities are stored in DRC’s Entity Table. The constraints of SQL entities are built similarly to those for PHP entities (Section IV).

**VII. Dangling Reference Detection Algorithm**

**A. Overview**

After collecting all declarations/references and constraints in the Entity Table, DRC matches each reference against declarations to check if it is dangling.

A reference and a declaration are considered signaturematched if they have the same signature (consisting of their name, type, and scope). As seen in Table III, a reference (e.g. in row 4) might signature-match with multiple declarations having different constraints (e.g. in rows 2 and 3) because they appear in different execution paths. To perform matching, DRC first combines all declarations having the same signature into a single meta-declaration with its constraint being the union of their constraints. The unioned constraint represents the program conditions of all possible executions in which the declaration appears. Thus, the unioned constraint for a declaration represents the program conditions for all configurations or all versions of the output client-side code in which that declaration occurs. For example, DRC combines the two declarations in rows 2 and 3 in Table III into one meta-declaration, whose constraint is \((\text{COND1} \lor \text{COND2}) \lor \neg \text{COND1}\). Let us call those meta-declarations as declarations for short.

A reference \(r\) and a declaration \(d\) are considered constraint-matched if in any program execution where \(r\) appears, \(d\) also appears. In other words, let \(C(r)\) and \(C(d)\) be the constraints of \(r\) and \(d\) respectively, then the condition for \(r\) and \(d\) to be constraint-matched can be expressed as \(C(r) \Rightarrow C(d)\) (1).

**Definition 5** (Dangling Reference). If a reference \(r\) does not both signature-match and constraint-match with any declaration \(d\), then \(r\) is dangling.

There are two cases of dangling reference: either there exists no declaration \(d\) with the same signature as \(r\); or such a declaration exists, but condition (1) is not satisfied with certain inputs and program states. That is, in the second case, there exist some program inputs and states that satisfy \(C(r)\) but not \(C(d)\). Equivalently, we have the following dangling condition:

\[ C(r) \land \neg C(d) \]

**B. Detailed Algorithm**

Figure 11 shows our algorithm to detect dangling references. First, DRC combines all entity declarations with the same signatures (line 2, Figure 11a). For two declarations having the same signature, the combination is done by updating one declaration with their combined (unioned) constraints (line 4 of Figure 11b), and removing the other from the EntityTable (line 5). After this step, the EntityTable contains the entity declarations with unique signatures.

Next, DRC searches for dangling references in the Entity Table. For each reference, it first checks if a declaration with the same signature exists (line 4, Figure 11a). If no such declaration is found, the reference is detected as dangling (line 8). Otherwise, DRC checks their constraints via the dangling condition to determine if the reference is dangling (lines 5-6).
Finding a solution to the constraint satisfaction problem (line 5, Figure 11a) is of high complexity as the variables in the constraints can be PHP numbers, strings, or objects, which can take on infinite numbers of values. To approximately solve that constraint satisfaction problem, DRC first transforms it into the boolean satisfiability problem. That is, it transforms the constraint expression into a boolean formula with the heuristic that the predicates are boolean variables whose values are either true or false, and are independent of each other. For example, the constraint \$lang == 'de' \&\& \$action == 'edit' is transformed into \( x \land \neg (x \lor y) \), assuming that \( x \) and \( y \) are independent and atomic boolean variables (‘atomic’ means that the variable itself is not a boolean formula of other variables). Thus, the number of possible combinations of values is far less than that of all arbitrary inputs of the PHP page.

In a PHP Web application, the number of options/configurations is not high. Therefore, although the number of operands in the transformed formula might be large, the number of atomic variables is often small. For example, the expression \( x \lor (\neg x \lor y) \land (x \lor \neg y) \) has five operands but only two atomic variables \( x \) and \( y \). In our current implementation, for a small number of variables, DRC tries different combinations of the variables’ values to decide if the formula is satisfiable with a set of input values. Otherwise, a constraint solver is used.

Let us illustrate our algorithm via Table III. First, DRC combines all declarations of \$groups_value (rows 2 and 3), and their combined constraint is \((\text{COND1} \land \text{COND2}) \lor \neg \text{COND1}\). The reference \$groups_value in row 4 (whose constraint is True) is signature-matched with the newly combined declaration. To check whether they are also signature-matched, DRC tests the satisfiability of the expression: True \&\& \neg (\text{COND1} \land \text{COND2} \lor \neg \text{COND1}), which is translated into True \&\& \neg (x \land y \lor \neg x). Since the expression evaluates to True with \( x = \text{True} \) and \( y = \text{False} \), the dangling condition holds. Thus, DRC detects that the PHP reference \$groups_value in row 4 is dangling in a configuration with \text{COND1}=\text{True} and \text{COND2}=\text{False}.

### VIII. Empirical Evaluation

Let us present our empirical studies to evaluate DRC’s accuracy in detecting dangling PHP and embedded references.

**(a) Detecting Dangling References**

```plaintext
1 function DetectDanglingReferences()
2     foreach Ref r in EntityTable
3          if exists Decl d such that Sig(d) == Sig(r)
4              if Satisfiable (Constraint(r) AND NOT Constraint(d))
5                  ReportDangling(r)
6         else
7             ReportDangling(r)
8
```

(b) Combining Entity Declarations

```plaintext
1 function CombineEntityDeclarations()
2     foreach Decl x in EntityTable
3         foreach Decl y in EntityTable such that Sig(y) == Sig(x)
4             Constraint(x) = Constraint(x) OR Constraint(y)
5     Remove y from EntityTable
6
```

Fig. 11. Dangling Reference Detection Algorithm

### A. Dangling Reference Detection Accuracy

In this experiment, we used the same data as in the study presented in Section II. During that study, from the commit logs and source code, three human subjects identified the cases of dangling PHP and embedded references reported by developers in those systems. We used them as the oracle. We ran DRC on those systems and manually checked the results. If the detected dangling cases were covered those in the oracle, we counted them as correct ones. If DRC reported a dangling case that was not in the oracle, we manually verified if it is a truly incorrect case or a newly discovered one (not yet reported). We then computed precision and recall. Precision is the ratio between the number of correctly detected cases over the total number of detected ones. Recall is the ratio between the number of correctly detected ones over the number of cases.

Table IV shows the results. Columns Corr, Incor, and Miss show the number of dangling cases that were correctly/incorrectly detected, and missed by DRC. Since an entity might have multiple references, in each table cell, two numbers are reported: the first one is the number of dangling entities and the second is the number of dangling references. As seen, DRC is accurate in dangling reference detection with an average of 89% recall (up to 100%) and 71% precision (up to 80%). Detection precision for dangling entities is higher. Interestingly, DRC discovered 83 not-yet reported cases (column New). We manually verified them as correct detection cases. DRC is also efficient. Detection time is typically less than two seconds per PHP file. Thus, our solution for matching constraints in the detection algorithm is practical. DRC’s demonstration and results are available at [32], [17]. Let us explain a few correctly detected cases in addition to those in Section II.

1) **PHP Dangling References in MRBS at Revision 590** (Figure 12): In addition to detecting the dangling reference \$typematch (not shown) reported in the commit log, DRC also found three other dangling ones that were not reported at revision 590: the three references to \$From_day, \$From_month, and \$From_year on line 368 are dangling since they are initialized only in the else branch of the if statement on line 316. As shown in the fix, this was corrected 210 revisions later by adding the variables’ initializations before line 316. Thus, DRC could have helped developers detect those dangling errors early.

2) **PHP Dangling Reference in ImpressCMS at Revision 3883** (Figure 13): Since the condition where \$cat_arr is defined
declarations/references occur only in PHP code. This requires DRC to have its specialized parsers on D-model to detect HTML/JS/SQL entities. Finally, the cross-language references in Web code and especially SQL entities add a new dimension.

Minamide [15]'s string analyzer approximates a PHP page's output via a context-free grammar. Wang et al. [22] use a similar string analyzer to approximate the output and identify the constant strings visible from a browser for translation. Wassermann and Su [23] combines tainted information flow with string analysis to detect cross-site scripting. Those works do not need to handle the constraints for multiple output versions and cross-language references as in DRC. Halfond and Orso [8] introduce a static analysis to identify parameter mismatches between the interfaces of Web components. They also build an approximation of all possible generated pages. Gould et al. [7] propose a static program analysis method to check the correctness of dynamically generated query strings.

PHP static analysis tools [33] (e.g., PHPLint [34]) detect only PHP undefined variable errors. They used static scoping, thus, cannot address dangling at run time in PHP. They cannot detect dangling references to undeclared entities in other languages (HTML/JS/SQL) and undefined PHP functions as in DRC. There are also compiling approaches for PHP (PyPy [30] and HipHop [31]). For JS, Saxena et al. [21] provides a framework for symbolic execution. Many authors introduced type inferencing in JS code [1], [9]. Several approaches were proposed to deal with eval in JS code [18], [10]. Clark and Tratt [5] give a formulation for embedded code.

There exist dynamic approaches for fault localization in a dynamic Web application. Apollo [2], [3] combines a variation of Tarantula [12] with the use of dynamic output mapping for PHP programs in which the PHP echo/print statements are recorded via an instrumented interpreter and are rated higher in suspiciousness. Clark et al. [4] propose a variation of Tarantula to localize faults in an SQL command embedded in JSP code.

There are several approaches for fault localization in traditional code. Many of them rely on changing program states at selected points in the execution of a failing test and observing the states’ differences or correlating them to failure [11], [25], [6], [14], [19], [26], [28]. Other approaches apply statistical metrics on program entities based on their coverage in passing/failing tests to assign suspiciousness scores [12], [13], [20].

X. Conclusions

This paper presents an empirical study on dangling PHP and embedded references, and DRC, a static analysis technique to detect them. DRC utilizes symbolic execution to produce D-model tree to approximate all versions of possible generated output. With a specialized parser for D-model, it extracts and validates references against declarations via our detection algorithm. Our evaluation shows that DRC can detect dangling references with high accuracy. It also revealed several not-yet-discovered defects caused by dangling references.

ACKNOWLEDGMENTS

This project is funded in part by NSF grants CCF-1018600, CNS-1223828, CCF-1320578, and CCF-1349153.
REFERENCES


[34] PHPLint. http://www.icosaedro.it/phplint/.