Variable Feature Usage Patterns in PHP

Mark Hills
East Carolina University, Greenville, NC, USA
mhills@cs.ecu.edu

***Abstract*—PHP allows the names of variables, classes, functions, methods, and properties to be given dynamically, as expressions that, when evaluated, return an identifier as a string. While this provides greater flexibility for programmers, it also makes PHP programs harder to precisely analyze and understand. In this paper we present a number of patterns designed to recognize idiomatic uses of these features that can be statically resolved to a precise set of possible names. We then evaluate these patterns across a corpus of 20 open-source systems totaling more than 3.7 million lines of PHP, showing how often these patterns occur in actual PHP code, demonstrating their effectiveness at statically determining the names that can be used at runtime, and exploring anti-patterns that indicate when the identifier computation is truly dynamic.**

I. Introduction

PHP is an imperative, object-oriented language focused on server-side application development. As of April 2015, it ranks 7th on the TIOBE programming community index,[1] and is used by 82 percent of all websites whose server-side language can be determined.[2] Designed to allow for the rapid construction of websites, PHP includes a number of dynamic language features used to simplify code, provide reflective capabilities, and support deferring configuration decisions to runtime.

An example of one such feature is *variable variables*. Instead of giving the name of a variable directly, it is given by an expression which should evaluate to a string containing the name of the variable. The actual variable to be accessed is then determined at runtime, based on the result of evaluating the expression providing the name. This provides a lightweight form of aliasing, and is often used to allow the same block of code to be applied to a number of different variables. At the same time, this makes it challenging to provide precise static analysis algorithms needed to support more advanced program analysis tasks and developer tools—without further analysis, a variable variable could refer to any variable in scope, including (if used in a `global` declaration) global variables.

In prior work [1] we showed that many occurrences of variable variables could actually be resolved to a limited set of names statically, just by inspecting the code. Many occurrences also fell into a small number of standard usage patterns. However, this prior work was not automated, but instead was based on reviewing how each variable variable was actually used in the program. The lack of automation makes it difficult to use these results in other analyses and tools, or to update them to take account of new systems or new releases of the analyzed systems. This prior work also focused just on variable variables, ignoring patterns of use of similar features for specifying the names of functions, methods, properties, and classes at runtime. Below, we refer to these generally as *variable features*, and specifically as, e.g., *variable functions* or *variable methods*.

The main contributions presented in this paper are as follows. First, taking advantage of our prior work, we have developed a number of patterns for detecting idiomatic uses of variable features in PHP programs and for resolving these uses to a set of possible names. Written using the Rascal programming language [2], [3], these patterns work over the ASTs of the PHP scripts, with more advanced patterns also taking advantage of control flow graphs and lightweight analysis algorithms for detecting value flow and reachability. Each of these patterns works generally across all variable features, instead of being designed specifically to recognize variable variables.

Second, to empirically determine how often these patterns actually occur in practice, we have applied them across a corpus of 20 open-source PHP systems. This corpus, made up of 31,624 files and 3,725,904 lines of PHP, includes a number of popular frameworks and systems including WordPress, Joomla, MediaWiki, and Symfony. Results are reported both in total and for groupings of similar patterns, providing insight into how each of the systems in the corpus uses variable features. Several anti-patterns, indicating that an occurrence is most likely unresolvable statically, are also presented; their effectiveness is measured by comparing detected occurrences with occurrences that are actually resolved using the patterns.

The rest of the paper is organized as follows. In Section II, we discuss PHP variable features in more depth, describing the various types of variable features available in the language and showing examples of how they are used. Section III then describes the corpus, tools, and research method applied to conduct this analysis. Following this, Section IV describes the patterns developed to detect idiomatic occurrences of variable features where these features can be statically resolved to precise sets of names as well as anti-patterns used to detect when this most likely is not possible. To determine how often these patterns and anti-patterns occur in actual open-source software, Section V presents the results from evaluating them using the corpus mentioned above. Finally, Section VI describes related work, and Section VII concludes. All software used in this paper, including the corpus used for the validation, is available for download at https://github.com/cwi-swat/php-analysis.

II. Overview

In PHP, a *variable feature* is a dynamic variant of a feature, like a variable, which normally uses an identifier but instead uses an expression to compute the name of the identifier. This allows the decision about which identifier to use to be deferred until runtime, providing a lightweight form of reflection that is

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[2] http://w3techs.com/technologies/details/pl-php/all/all

TABLE I. VARIABLE FEATURES IN PHP.

Language Feature	Expression Allowed For
Variable	Variable Name
Object Creation/Instantiation	Class Name
Function Call	Function Name
Method Call	Method Name
Static Method Call	Class Name, Method Name
Property Fetch/Reference	Property Name
Static Property Reference	Class Name, Property Name
Class Constant Reference	Class Name

used to support runtime configuration and generic algorithms while minimizing code duplication.

Table I shows the variable features available in PHP. The first column shows the language feature where an expression can be used in place of an identifier, while the second column shows where in this feature these expressions can appear. For instance, an expression can be used to compute the name of the variable when variables are used, or the name of a class in a `new` expression, a static method call, or a class constant reference. Note that, in PHP, variables are prefixed with `$`, like `$var`. If an expression is used in a position where an identifier would normally be expected, such as following the `$` used for a variable, this indicates that the expression should be evaluated to compute the identifier name, so `$$var` computes the value of `$var` and uses this as the variable identifier. The expression after `$` can also be wrapped in braces, like `${$var . 'local'}`, when a more complex expression (here, looking up the value of `$var` and concatenating `'local'`) is used.

An example of each of these features is shown in Figure 1. All of these examples are from the corpus: the first five examples are from MediaWiki 1.24.1; example six is from the Zend Framework 2.3.7; examples seven and eight are from Moodle 2.8.5; example nine is from CakePHP 2.6.3; and example ten is from phpBB 3.1.3. Each example gives the feature being shown as well as the path to the file containing this occurrence and the line number in the file where the occurrence can be found. In order, the examples show the following:

- EX1 The name of the variable to use is computed by evaluating `$var`.
- EX2 The function to call is computed by evaluating `$strcmp`.
- EX3 The method to call is computed by evaluating `$method`.
- EX4 The class used to create a new object instance is computed by evaluating `$class`.
- EX5 The property to use is computed by evaluating `$o`.
- EX6 The class containing the class constant being used is computed by evaluating `$color`.
- EX7 The static method to call is computed by evaluating `$method` (the class name is also computed, see the next example).
- EX8 The class containing the static method being called is computed by `$class`.
- EX9 The static property to use is computed by `$name`.
- EX10 The class containing the static property to look up is computed by `$type` (this example also has a variable static property).

Figure 2 shows how the first example is actually used,

```
1  // EX1. Variable variable,
2  //    /includes/Sanitizer.php, line 427
3  $$var = array_flip( $$var );
4
5  // EX2. Variable function call,
6  //    /includes/utils/StringUtils.php, line 187
7  $strcmp( $endDelim,
8          substr( $subject, $tokenOffset, $endLength ) )
9
10 // EX3. Variable method call,
11 //    /maintenance/fileOpPerfTest.php, line 139
12 $backend->$method( $ops5, $opts )
13
14 // EX4. Variable object instantiation,
15 //    /includes/WebRequest.php, line 834
16 $this->response = new $class();
17
18 // EX5. Variable property lookup,
19 //    /includes/HttpFunctions.php, line 259
20 $this->$o = $options[$o];
21
22 // EX6. Variable class constant lookup,
23 //    /library/Zend/Console/Adapter/Posix.php, line 388
24 $color::BACKGROUND
25
26 // EX7. Variable static method call,
27 //    /cache/tests/fixtures/lib.php, line 528
28 $class::$method($definition)
29
30 // EX8. Variable target for static method call,
31 //    /cache/classes/factory.php, line 352
32 $class::config_file_exists()
33
34 // EX9. Variable static property lookup,
35 //    /lib/Cake/Utility/CakeTime.php, line 118
36 return self::${$name};
37
38 // EX10. Variable target for static property lookup,
39 //     /notification/manager.php, line 569
40 $type::$notification_option
```

Fig. 1. Examples of Variable Features in PHP.

putting it into context by showing the surrounding code (slightly reformatted to fit better). Here, an array of string literals is created and assigned to variable `$vars`. The `foreach` loop then takes each element of `$vars` in turn and assigns it to variable `$var`. This has the effect of running the body of the loop on each of the variables named by the string literals in the array. In PHP, arrays are actually dictionaries, with each value associated with a key— standard arrays without explicit keys are associated with numeric keys indicating their index. The body takes the arrays `htmlpairsStatic`, `htmlsingle`, etc., and "flips" them, inverting each array so that the keys become the values and the values the keys. Although dynamic, the behavior here is essentially static, with the names the variable can reference given directly in the code. Patterns such as this are common idioms for variable feature usage, so recognizing occurrences of these patterns is important for properly understanding the code. Section IV describes these patterns, and how they are recognized, in more detail.

```
1 // MediaWiki, /includes/Sanitizer.php, lines 424-428
2 $vars = array( 'htmlpairsStatic', 'htmlsingle',
3   'htmlsingleonly', 'htmlnest',
4   'tabletags', 'htmllist', 'listtags',
5   'htmlsingleallowed', 'htmlelementsStatic' );
6 foreach ( $vars as $var ) {
7   $$var = array_flip( $$var );
8 }
```

Fig. 2. Variable Variables with a foreach Loop, Matching Pattern LP-2.

TABLE II. THE PHP CORPUS.

System	Version	PHP	Release Date	File Count	SLOC	Description
CakePHP	2.6.3	5.2.8	3/16/15	671	155,320	Application Framework
CodeIgniter	3.0.0	5.2.4	3/31/15	198	29,169	Application Framework
Doctrine ORM	2.4.7	5.3.2	12/16/14	763	64,168	Object-Relational Mapping
Drupal	7.35	5.2.4	3/18/15	276	89,658	CMS
Gallery	3.0.9	5.2.3	6/28/13	505	39,087	Photo Management
Joomla	3.4.1	5.3.10	3/21/15	2,333	247,426	CMS
Kohana	3.3.3.1	5.3.3	12/10/14	482	30,207	Application Framework
Magento	1.9.1.0	5.2.0	11/24/14	8,248	652,116	Online Retail
MediaWiki	1.24.1	5.3.2	12/17/14	1,710	321,928	Wiki
Moodle	2.8.5	5.4.4	3/10/15	7,515	1,027,999	Online Learning
osCommerce	2.3.4	4.0.0	6/5/14	698	60,003	Online Retail
PEAR	1.9.5	4.4.0	7/12/14	74	31,283	Component Framework
phpBB	3.1.3	5.3.3	2/1/15	740	182,969	Bulletin Board
phpMyAdmin	4.3.13	5.3.0	3/29/15	487	149,765	Database Administration
SilverStripe	3.1.2	5.3.2	10/22/13	572	92,216	CMS
Smarty	3.1.21	5.2.0	10/18/2014	126	16,266	Template Engine
Squirrel Mail	1.4.22	4.1.0	7/12/11	276	36,082	Webmail
Symfony	2.6.5	5.3.3	3/17/15	3,083	200,219	Application Framework
WordPress	4.1.1	5.2.4	2/18/15	476	141,399	Blog
The Zend Framework	2.3.7	5.3.23	3/12/15	2,391	158,624	Application Framework

The PHP versions listed in column PHP are the minimum required versions. The File Count includes files with a .php or an .inc extension. In total there are 20 systems consisting of 31,624 files with 3,725,904 total lines of source.

III. CORPUS AND RESEARCH METHOD

In this section we first introduce the corpus used in this paper. We then discuss our research method at a high level, describing how the experiment was conducted.

A. Corpus

The corpus used in this paper includes 20 large open-source PHP systems. These systems were selected based on popularity rankings provided by Black Duck's Open Hub site,[3] a site that tracks open-source projects. The systems are shown in Table II, with versions based on what was current at the end of March 2015. Systems were initially selected mainly based on this ranking, although in some cases we skipped systems if we already had several, more popular systems of the same type in the corpus, ensuring diversity in the types of systems we are analyzing. We used popularity, instead of actual number of downloads or installed sites, since we have no way to accurately compute the number of downloads or installations. In total, the corpus consists of $31,624$ PHP files with $3,725,904$ lines of PHP source, counted using the cloc [4] tool.

When we initially assembled the corpus for our earlier work [1] we focused on larger, more widely used systems for two reasons. First, these systems are more likely to benefit from improved tools and analysis techniques than smaller systems, and second, larger systems should provide more diversity in which PHP features are used and in how they are used, providing more usage patterns like those we discuss in Section IV. We also had the benefit of having looked specifically at variable variables in this earlier work, categorizing them generally into how many could be resolved to specific sets of identifiers by code inspection versus how many appeared to actually be dynamic.[4] This paper extends this work by automating this process based on common usage patterns and by looking at all variable features, not just variable variables.

B. Research Method

PHP AiR, a framework for PHP Analysis in Rascal [5], is used to perform all the analysis used to compute the results in this paper. Rascal [2], [3] is a meta-programming language for source code analysis and transformation. PHP AiR is written in Rascal, and makes heavy use of Rascal features such as pattern matching, tree traversals, and relations to extract information from PHP Abstract Syntax Trees using the patterns described in Section IV. PHP AiR includes support for parsing PHP scripts, building and traversing control flow graphs, and performing simplifications on PHP expressions using techniques such as algebraic simplification and function call simulation, all of which are used in the pattern recognizers described in Section IV. Using Rascal ensures that the analysis itself is fully scripted and reproducible, including the generation of the LaTeX used for the tables and `pgfplots` figures in this paper. All code is available online at https://github.com/cwi-swat/php-analysis.

The parser used inside PHP AiR is our fork[5] of an open-source PHP parser[6] based on the grammar used inside the Zend Engine, the scripting engine for PHP. This parser generates ASTs as terms formed over Rascal's algebraic datatypes. We have opted to reuse an existing PHP parser instead of creating one in Rascal since this makes it easier to stay compatible with changes to the PHP language as it evolves.

To conduct the experiment, first all occurrences of variable features in the corpus were identified using pattern matching to detect all places where an expression was used in place of an identifier. Each pattern recognizer was then run over all occurrences that had not already been resolved by an earlier pattern (the patterns are generally distinct, so this is mainly

[3] https://www.openhub.net/tags?names=php, formerly known as Ohloh.

[4] Note that this earlier work did not include Magento, which was added to the corpus later.

[5] https://github.com/cwi-swat/PHP-Parser

[6] https://github.com/nikic/PHP-Parser/

TABLE III. LOOP PATTERNS.

Pattern ID	Description
LP-1	Foreach iterates directly over array of string literals, value variable used directly to provide identifier
LP-2	Foreach iterates over array of string literals assigned to array variable, value variable used directly to provide identifier
LP-3	Foreach iterates directly over array of string literals, key variable used directly to provide identifier
LP-4	Foreach iterates over array of string literals assigned to array variable, key variable used directly to provide identifier
LP-5	Foreach iterates directly over array of string literals, intermediate uses value variable to compute new string, intermediate then used to provide identifier
LP-6	Foreach iterates over array of string literals assigned to array variable, intermediate uses value variable to compute new string, intermediate then used to provide identifier
LP-7	Foreach iterates directly over array of string literals, intermediate uses key variable to compute new string, intermediate then used to provide identifier
LP-8	Foreach iterates over array of string literals assigned to array variable, intermediate uses key variable to compute new string, intermediate then used to provide identifier
LP-9	Foreach iterates directly over array of string literals, value variable used as part of expression directly in occurrence to compute identifier
LP-10	Foreach iterates over array of string literals assigned to array variable, value variable used as part of expression directly in occurrence to compute identifier
LP-11	Foreach iterates directly over array of string literals, key variable used as part of expression directly in occurrence to compute identifier
LP-12	Foreach iterates over array of string literals assigned to array variable, key variable used as part of expression directly in occurrence to compute identifier
LP-13	For iterates over numeric range, string literal and loop index variable used as part of expression directly in occurrence to compute identifier
LP-14	For iterates over numeric range, intermediate combines string literal with loop index variable, intermediate then used to provide identifier

done to save time). The result of running a recognizer on a specific system is a summary by feature indicating how many occurrences of this pattern can be resolved to a precise set of names, the names derived for a solved occurrence, and the occurrences that matched the general structure of the pattern but could not be resolved. The overall result for a specific pattern is then a map from the system name to these summaries, with a total of 20 summaries per pattern. This is then serialized, making it easy to load the results later for further analysis and to generate the tables and figures in the paper. The numbers shown in this paper in Section V, giving the results of the patterns for the corpus, are directly based on these summaries.

IV. RESOLVING VARIABLE FEATURES

In this section we present common variable feature usage patterns in PHP grouped into three categories: loop patterns, assignment patterns, and flow patterns. For each of these three categories we describe each pattern and show one or more examples of code that matches patterns in that category. Following this, we also discuss several anti-patterns which indicate that a given use of a variable feature is truly dynamic.

A. Loop Patterns

One common usage pattern for variable features is to iterate over an array of identifier names using a `foreach` loop, or, less commonly, to derive identifier names based on a string and a loop index variable in a `for` loop (`$var1`, `$var2`, etc). To detect these cases, we have developed 14 loop patterns, listed in Table III. Patterns LP-1 through LP-12 all involve `foreach` loops, while patterns LP-13 and LP-14 use `for` loops.

All the patterns that match `foreach` loops are quite similar, varying in three major ways. First, in some patterns an array of scalar strings is given directly inside the `foreach` construct, like what is seen in Figure 3. This is done in all the patterns that specify the `foreach` iterates directly over an array of string literals: LP-1, LP-3, LP-5, LP-7, LP-9, and LP-11. In the other `foreach` patterns— LP-2, LP-4, LP-6, LP-8, LP-10, and LP-12—the array is assigned into a variable first, called an array variable in Table III, and the `foreach` iterates over that. Second, since PHP arrays are actually key/value mappings, and since the foreach can iterate over either just the values (see Figure 2) or the key/value mappings (see Figure 3), in some of the patterns (LP-1, LP-2, LP-5, LP-6, LP-9, LP-10) the value is used in the identifier computation, while in the other `foreach` patterns (LP-3, LP-4, LP-7, LP-8, LP-11, LP-12) the key is used in the identifier computation. Third, in some of the patterns (LP-1 through LP-4) the key or value is used directly, while in others (LP-5 through LP-12) some computation is performed first, either by saving this computation into an intermediate variable that provides the identifier name (LP-5 through LP-8) or by performing the computation directly inside the occurrence of the variable feature (LP-9 through LP-12), with various simplifications such as simulating common operations and functions performed to attempt to reduce the expression to a set of string literals, one for each possible name.

Two examples of patterns matching `foreach` loops are shown in Figures 2 and 3. The first is an example of LP-2: an array of string literals is assigned to variable `$vars`, which is then used in a `foreach` loop, with the value at each iteration assigned to `$var` and used to directly provide the identifier to pass to `array_flip`. The second, with some parts of the code elided, is an example of LP-7: an array with string literals for both keys and values is given directly in the `foreach` loop, and this key is used to compute a class name, `$tag_class`, that is used in a `new` expression.

Patterns LP-13 and LP-14 instead match `for` loops, with the identifier computed as a combination of string literals and the loop index variable. The main difference between LP-13

```
1 // WordPress, /wp-includes/ID3/getid3.php, lines 345-358
2 foreach (array('id3v2'=>'id3v2', ...)
3         as $tag_name => $tag_key) {
4         ...
5         $tag_class = 'getid3_'.$tag_name;
6         $tag = new $tag_class($this);
7         ...
8 }
```

Fig. 3. Loop Matching Pattern LP-7.

```
1 // SquirrelMail,/src/options_highlight.php,lines 339-341
2     for ($i=0; $i < 14; $i++) {
3         ${"selected".$i} = '';
4     }
```

Fig. 4. Loop Matching Pattern LP-13.

TABLE IV. ASSIGNMENT PATTERNS.

Pattern ID	Description
AP-1	String literals assigned into variable, variable used directly to provide identifier
AP-2	String literals assigned into variable conditionally based on ternary expression, variable used directly to provide identifier
AP-3	Array of string literals assigned into variable, array indexed into to provide identifier
AP-4	String literals assigned into variable, variable used as part of expression in occurrence to compute identifier

TABLE V. FLOW PATTERNS.

Pattern ID	Description
FP-1	Ternary used in expression in occurrence to compute identifier
FP-2	Conditional compares variable to string literal values, variable used directly to find identifier
FP-3	Switch/case switches on variable with literal cases, variable used directly to find identifier
FP-4	Conditional compares variable to string literal values, variable used as part of expression in occurrence to compute identifier
FP-5	Switch/case switches on variable with literal cases, variable used as part of expression in occurrence to compute identifier

and LP-14 is that, in LP-13, the computation of the identifier takes place directly in the variable feature occurrence, while in LP-14 it is assigned to an intermediate variable first which is then used, without further computation, as the identifier name. Figure 4 shows an example matching pattern LP-13, with a sequence of variables `$selected0`, `$selected1`, through `$selected13` assigned the empty string.

B. Assignment Patterns

Another common usage pattern for variable features is to use one or more assignment statements to assign a string literal into a variable that is then used to compute the identifier, either directly, using only the value assigned into the variable, or as part of an expression that returns a string. To detect these cases, we have developed 4 assignment patterns, listed in Table IV.

Patterns AP-1 and AP-4 both match assignments of string literals into single variables. In AP-1, this variable is used directly in the variable feature to provide the identifier to be used at runtime, while in AP-4 this variable is used as part of a computation directly in the variable feature that yields the identifier as a string. AP-2 handles the special case where a ternary conditional expression is used to select between literal values to assign into a variable, which is then used directly to provide the identifier. Finally, AP-3 handles the case where an array of string literals is assigned into a variable, and this variable is then indexed into without being used in a `for` or `foreach` statement in a way that would match the loop patterns given above (i.e., in a way where the loop index variable does not directly contribute to the name). At runtime, the identifier will be given by one of the literals in the array.

All of the assignment patterns share two important features. First, all require an explicit assignment of a string literal into the variable used in the variable feature occurrence on all paths from the start of the script, function, or method to the occurrence. Second, other assignments of string literals are allowed, but there is no effort to detect when one assignment *kills* another, meaning the analysis is essentially flow-insensitive once we know all paths that reach the use in the variable feature have a definite assignment. The control-flow graph construction functionality in PHP AiR is used to enable this analysis.

```
1  // WordPress,/wp-includes/class-wp-customize-setting.php,
2  // lines 334-361
3  switch( $this->type ) {
4    case 'theme_mod' :
5      $function = 'get_theme_mod';
6      break;
7    case 'option' :
8      $function = 'get_option';
9      break;
10   default :
11     ...
12     return ...
13 }
14 // Handle non-array value
15 if ( empty( $this->id_data[ 'keys' ] ) )
16  return $function($this->id_data['base'],$this->default);
```

Fig. 5. Assignment Matching Pattern AP-1.

Figure 5 shows an example matching pattern AP-1, with some parts of the code elided. The variable feature is a variable function call, shown on the last line with `$function` used to provide the name of the function. Above this, a switch statement includes two case which both assign a string literal to variable `$function`. The `default` case includes a `return` statement, so control that goes through the `default` case will never reach the last line, allowing the analysis to determine that all reaching paths have a definite assignment of a string literal to `$function`. The result is that either function `get_theme_mod` or `get_option` could be invoked.

C. Flow Patterns

The final usage pattern for variable features shown in this paper is where non-looping control flow is used to either provide the value for the identifier or, through comparisons, to indicate which values are being used. To detect these cases, we have developed 5 flow patterns, listed in Table V.

The first flow pattern, FP-1, matches cases where a ternary conditional expression is given directly as the expression used to compute the identifier, with this expression used to select different string literals. FP-2 and FP-4 both match cases using conditional statements. In FP-2, the identifier is provided directly by a variable for an occurrence appearing inside an `if` or `elseif` branch (PHP conditional statements include a single `if` clause, zero or more `elseif` clauses,

```
1  // WordPress, /wp-includes/capabilities.php,
2  // lines 1054-1332
3  switch ( $cap ) {
4          ...
5          case 'delete_post':
6          case 'delete_page':
7                  ...
8                  $caps[] = $post_type->cap->$cap;
9                  ...
10         }
11         ...
12 }
```

Fig. 6. Flow Matching Pattern FP-3.

TABLE VI. ANTI-PATTERNS.

Pattern ID	Description
ANTI-1	Variable used in identifier computation is a function or method parameter
ANTI-2	Variable used in identifier computation is result of function or method call
ANTI-3	Variable used in identifier computation is global

and an optional `else`) where that same branch is selected by comparing the variable to a literal string. FP-4 is the same, except the variable is not used directly, but is instead used inside a computation that yields a string literal. Finally, FP-3 and FP-5 both match scenarios using `switch`/`case` statements. In FP-3, the identifier is provided by a variable that is also used as the `switch` expression, with possible values of this variable given using string literals for each case. FP-5 is the same except, as with FP-4, this variable is used inside a computation that yields a string literal. Both FP-3 and FP-5 account for possible fall-through, working backwards from the point of use to see which cases could reach that point.

Figure 6 shows an example matching pattern FP-3, with some parts of the code elided. Here, variable `$cap` is used in the `switch`, with the various cases indicating possible values. Two cases, `'delete_post'` and `'delete_page'`, then reach the last line of code shown except for the closing brace, where `$cap` is used to provide the name of a method called in expression `$post_type->cap->$cap`. Based on the string literals assigned to the cases, this method should be either `delete_post` or `delete_page`.

D. Variable Feature Anti-Patterns

The patterns described above are not exhaustive: there are uses that do not match any pattern, and some uses that cannot be statically resolved at all but are truly dynamic. The variable features in PHP are often used to support runtime configuration, for instance by allowing additional fields representing user-defined metadata to be added to objects representing blog posts. In these cases the identifiers used at runtime are based on database settings or configuration files, and vary across installed systems and even over time for the same system.

To detect situations where this is the case, we have initially defined three anti-patterns, listed in Table VI. Our use of "anti" does not connote bad programming practice, but that a matching occurrence likely cannot be resolved statically. All three patterns represent cases where information used to compute the identifiers for a given occurrence come in from outside of the current script, function, or method. In ANTI-1, a variable used in the computation is a parameter to the current function or method; in ANTI-2, the variable receives a value from a function or method call; and in ANTI-3, the variable is declared to be a global variable using a `global` declaration.

V. EVALUATION

In this section we describe the results of our evaluation of the patterns and anti-patterns described in Section IV. We also discuss threats to the validity of our results.

A. Variable Feature Usage

Table VII summarizes occurrences of variable features across the entire corpus. The first two columns show the name of the system and the number of files with `.php` or `.inc` extensions (used to represent include files). Following this there are five column pairs containing statistics on the most commonly used variable features: variable variables, function calls, method calls, property fetches, and instantiations, respectively. Each pair includes the number of files that contain the specific feature as well as the number of uses across all the files in total. The last three columns then present statistics for all uses of variable features, including those not shown in the column pairs. Included are the total number of files where a variable feature occurs, the total number of occurrences, and the Gini coefficient for the distribution of variable feature occurrences across those files containing at least one. The Gini, a real number between 0 and 1, measures how evenly spread these features are through all the files, with higher scores indicating that variable features tend to occur in clusters. While the figures in Table VII show that variable features occur in a small percentage of files (roughly 7.73%), our earlier work [1] showed that variable features often occur in files that are included into other files, at least doubling the chance of encountering one in a script being analyzed. We also believe the fact that they are clustered makes resolution more important for understanding and analyzing specific files and subsystems where they occur in larger numbers.

B. Effectiveness of the Loop Patterns

Table VIII summarizes the results of running the loop patterns listed in Table III. As with Table VII, the first column shows the name of the system. The remaining columns show how many occurrences of the loop patterns were either resolved or unresolved for specific variable features—variable variables, function calls, method calls, property fetches, and instantiations—and, with the last two columns, across all variable features, including those not shown. These counts are summed across all 14 loop patterns, and focus just on those cases where the pattern matched (e.g., where the `foreach` variable is used directly to name an identifier, as in pattern LP-1) and either could be resolved (e.g., again for LP-1, where all the array values are literal strings and where the `foreach` variable is not otherwise modified) or could not be resolved (e.g., again for LP-1, but where the array values are not literal strings or where the `foreach` variable may be modified before it is used in a way we do not, or cannot, statically model).

The results show that uses of variable features with `for` and `foreach` loops are very common in the corpus: 168 (40 resolved, 128 unresolved) of the 490 occurrences in CakePHP are with `for` and `foreach` loops, as are 1,123 of the 2,501 occurrences in Moodle. This isn't always the case, though. For instance, in osCommerce only 14 of the 175 occurrences involve `for` and `foreach` loops, while in Symfony this is only 23 of 268. The results also show that the loop patterns are more effective at resolving names related to variable variables and properties than to the other features. In CakePHP 13 of the 14 occurrences of variable variables that involve `for` or `foreach` loops are resolved, and in CodeIgniter all 12 occurrences are resolvable. The numbers for variable properties are not as encouraging, but a significant number are still resolvable, even with higher unresolved counts: 158 of 943 in Moodle, 24 of 62 in WordPress, and 23 of 144 in CakePHP. However, for function calls, only 2 in the entire corpus can be resolved with loop patterns, and for method calls only 20 can be resolved. At the same time, loop patterns for variable functions and methods are

TABLE VII. PHP VARIABLE FEATURES.

System	Files	PHP Variable Features												
		Variables		Function Calls		Method Calls		Property Fetches		Instantiations		All		
		Files	Uses	Files	Uses	Files	Uses	Files	Uses	Files	Uses	Files	Uses	Gini
CakePHP	671	5	14	0	0	15	25	56	327	42	110	92	490	0.60
CodeIgniter	198	5	12	7	15	11	25	26	82	15	23	42	157	0.46
DoctrineORM	763	0	0	2	3	7	13	3	85	15	29	25	131	0.70
Drupal	276	1	1	33	377	2	3	20	96	13	25	50	502	0.73
Gallery	505	3	7	3	7	6	14	26	96	13	19	47	155	0.52
Joomla	2,333	2	3	7	12	24	37	151	648	85	210	241	940	0.57
Kohana	482	3	7	4	12	7	14	7	17	13	14	30	69	0.42
Magento	8,248	11	32	13	36	129	174	138	488	199	309	428	1,043	0.46
MediaWiki	1,710	4	8	12	26	15	25	52	106	74	90	143	255	0.31
Moodle	7,515	16	36	68	200	67	109	357	1,624	209	371	631	2,501	0.54
osCommerce	698	24	119	1	2	0	0	13	23	17	29	52	175	0.49
PEAR	74	1	1	1	2	7	16	2	7	16	22	23	48	0.38
phpBB	740	15	61	4	27	4	5	5	12	25	37	51	148	0.48
phpMyAdmin	487	8	30	11	33	6	12	6	13	15	18	37	112	0.52
SilverStripe	572	2	2	14	20	41	240	50	165	56	154	120	607	0.68
Smarty	126	10	40	6	12	6	22	5	12	12	22	32	108	0.44
SquirrelMail	276	5	24	13	24	0	0	2	3	0	0	18	51	0.47
Symfony	3,083	0	0	29	57	21	93	14	53	28	36	94	268	0.56
WordPress	476	1	1	7	25	5	5	56	196	17	18	77	246	0.43
ZendFramework	2,391	2	4	52	219	58	84	44	98	71	99	213	548	0.49

TABLE VIII. RESULTS OF MATCHING LOOP PATTERNS LP-1 THROUGH LP-14.

System	Resolved and Unresolved Variable Features, Loop Patterns LP-1 Through LP-14											
	Variables		Function Calls		Method Calls		Property Fetches		Instantiations		All	
	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved
CakePHP	13	1	0	0	4	1	23	121	0	3	40	128
CodeIgniter	12	0	0	3	2	7	7	32	0	0	21	42
DoctrineORM	0	0	0	0	1	3	0	4	0	1	1	8
Drupal	1	0	0	36	0	1	11	26	0	1	12	64
Gallery	2	5	0	0	0	6	19	27	0	1	22	41
Joomla	1	0	0	1	0	2	4	145	0	6	5	156
Kohana	0	5	0	0	0	6	1	5	0	0	1	16
Magento	18	6	0	3	2	54	9	155	0	16	29	236
MediaWiki	4	2	0	3	0	3	17	36	0	2	21	46
Moodle	11	5	2	38	0	26	158	785	1	57	172	951
osCommerce	0	3	0	0	0	0	3	2	0	6	3	11
PEAR	1	0	0	1	6	7	0	5	0	2	7	15
phpBB	28	23	0	0	0	0	0	3	0	1	29	30
phpMyAdmin	1	4	0	3	0	0	0	9	0	0	1	16
SilverStripe	1	1	0	3	0	25	5	58	0	10	6	106
Smarty	8	30	0	0	0	0	0	0	0	0	8	30
SquirrelMail	6	14	0	4	0	0	0	0	0	0	6	18
Symfony	0	0	0	8	3	5	5	2	0	0	8	15
WordPress	0	1	0	0	0	0	24	38	1	1	25	40
ZendFramework	0	2	0	4	2	41	3	37	0	8	5	94

uncommon across much of the corpus. This intuitively makes sense, as a developer is more likely to want to repeat similar computations for a group of variables or object properties than to repeatedly apply a collection of functions or methods to the same values or variables.

C. Effectiveness of the Assignment Patterns

Table IX summarizes the results of running the assignment patterns listed in Table IV. The table is formatted identically to that for Table VIII, with unresolved representing those cases where there is a definite assignment to a variable used to compute the identifier in a variable feature occurrence but where this assigned value is not a string literal, even after applying simplifications like those discussed above.

The results show that uses of variable features relying on names that are not controlled by a `foreach` or `for` loop are also very common. The results also show that assignment patterns are much more effective at resolving the names used by variable function and method calls than were the loop patterns. 9 of the 14 variable function occurrences in CodeIgniter that could be resolved are resolved, while all 7 in Gallery are resolved and 20 of 35 in Magento are resolved. For variable method calls, 144 of the 208 occurrences in SilverStripe are resolved, 2 of the 3 in Drupal, and 6 of the 19 in MediaWiki. Not all systems give such good results, though: only 21 of the 150 occurrences are resolved in Magento, and none of the 30 in Joomla are resolved.

D. Effectiveness of the Flow Patterns

Table X summarizes the results of running the flow patterns listed in Table V. The table is formatted identically to that for Table VIII, with unresolved representing those cases where there is a matching control flow case (e.g., a variable is used to compute the identifier in a variable feature occurrence, and this variable is also used in a condition or `switch`) but it is not possible to compute a string literal in the comparison or for the cases. Note that the unresolved figures may be overly pessimistic, since they also include occurrences where a variable occurs in a condition or switch but the intent is not to directly compare it to possible identifier values.

TABLE IX. RESULTS OF MATCHING ASSIGNMENT PATTERNS AP-1 THROUGH AP-4.

System	Resolved and Unresolved Variable Features, Assignment Patterns AP-1 Through AP-4											
	Variables		Function Calls		Method Calls		Property Fetches		Instantiations		All	
	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved
CakePHP	0	1	0	0	0	16	0	135	1	36	1	198
CodeIgniter	0	0	9	5	0	20	2	42	0	22	11	89
DoctrineORM	0	0	0	2	5	3	0	4	1	19	6	29
Drupal	0	0	3	359	2	1	0	51	1	22	6	433
Gallery	0	5	7	0	0	7	0	44	0	19	8	81
Joomla	0	2	4	8	0	30	0	471	10	107	14	641
Kohana	0	4	0	8	4	5	0	11	0	14	4	43
Magento	6	8	20	15	21	129	5	269	16	263	68	688
MediaWiki	1	0	1	23	6	13	0	43	2	74	10	153
Moodle	0	18	27	132	5	81	17	720	26	294	80	1,372
osCommerce	0	31	0	2	0	0	0	19	1	23	1	75
PEAR	0	0	1	1	0	10	0	2	2	16	3	29
phpBB	0	23	2	16	3	1	4	6	0	33	9	81
phpMyAdmin	0	14	3	10	7	4	0	6	0	17	10	55
SilverStripe	0	1	0	14	144	64	0	100	2	74	146	268
Smarty	0	6	0	10	0	5	0	4	0	10	0	35
SquirrelMail	0	5	1	5	0	0	0	2	0	0	1	12
Symfony	0	0	0	46	0	24	0	25	0	29	0	152
WordPress	0	1	5	9	0	5	3	72	0	13	8	101
ZendFramework	0	4	0	213	7	67	0	54	3	76	10	455

TABLE X. RESULTS OF MATCHING FLOW PATTERNS FP-1 THROUGH FP-5.

System	Resolved and Unresolved Variable Features, Flow Patterns FP-1 Through FP-5											
	Variables		Function Calls		Method Calls		Property Fetches		Instantiations		All	
	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved	Resolved	Unresolved
CakePHP	0	0	0	0	2	7	10	141	0	11	14	163
CodeIgniter	0	0	0	4	0	10	0	39	0	1	0	54
DoctrineORM	0	0	0	0	0	0	0	6	0	2	0	8
Drupal	0	0	0	51	0	0	0	36	0	4	0	91
Gallery	0	2	0	0	2	7	3	42	0	0	6	60
Joomla	0	0	0	3	10	16	61	247	0	68	75	341
Kohana	0	4	0	0	0	2	2	4	0	1	2	11
Magento	2	0	0	5	6	82	14	261	2	48	24	397
MediaWiki	0	0	0	5	0	2	5	32	1	8	6	47
Moodle	1	4	0	107	2	57	28	643	0	45	31	906
osCommerce	0	53	0	0	0	0	0	6	0	12	0	71
PEAR	0	0	0	1	0	8	4	3	0	1	4	13
phpBB	4	9	0	3	1	1	0	4	0	6	5	26
phpMyAdmin	2	15	0	11	0	0	0	6	0	1	2	36
SilverStripe	0	0	0	9	0	47	0	66	0	34	0	172
Smarty	26	0	0	9	0	18	8	0	0	8	34	35
SquirrelMail	0	10	0	7	0	0	0	2	0	0	0	19
Symfony	0	0	0	13	0	11	0	9	0	8	0	43
WordPress	0	0	0	12	1	1	6	57	0	6	7	76
ZendFramework	2	0	0	48	6	40	0	32	0	37	8	158

The results show that uses of variable features that interact with ternary conditional expressions, conditionals, and `switch` statements are also fairly common, although not as much for variable variables as for the other common variable features (especially variable properties). In several cases resolution works quite well: in Smarty, the Zend Framework, and Magento, all the variable variables occurring inside these constructs are resolved (although for all but the first the number is quite small), as well as all the variable properties in Smarty, and in Joomla 10 of the 26 variable functions can be resolved. In many cases, though, occurrences of these patterns cannot be resolved: none of the 107 variable functions in Moodle are resolved, and neither are any of the 45 instantiations.

E. Overall Effectiveness

Table XI summarizes the results across all patterns. For each variable feature, the Resolved column sums all resolved features, while the Uses column contains a count of all occurrences of that feature in the given system (and is the same number as was shown in Table VII). Overall, except for certain outliers (e.g., osCommerce), variable variables are often used in static ways, mainly to eliminate code duplication, and many of these cases can be detected by our patterns. 40.8% of the variable variable occurrences are currently resolved, and this reaches 60.5% if we focus just on systems that require some version of PHP 5. The overall percentage that are resolved is 13.3%, with a higher percentage for variable methods (29.5%) and lower percentages for variable functions (8.9%), property fetches (11.6%), and object instantiations (6.2%). While certain systems show higher rates of success than others, in many cases the unresolved features are either truly dynamic or require more sophisticated analysis techniques than those employed in Section IV.

F. Anti-Patterns

The challenge in resolving many of the variable feature occurrences in the corpus leads to a natural question: are these uses truly dynamic, or would a stronger analysis, not focused on detecting usage patterns but just on resolving the identifiers to sets of strings, be more effective? Table XII shows the results of running the anti-patterns described in Table VI. Similar to the above tables, the Detected column shows how many occurrences of the anti-patterns were detected for that given feature (or, for the second to last column, all features) in that system, while the Resolved column shows how many of these occurrences were actually resolved. Good anti-patterns will have very low resolved counts, since the anti-pattern is trying to detect cases where the feature is most likely used in a dynamic way and thus cannot be resolved.

TABLE XI. OVERALL RESULTS OF MATCHING ALL PATTERNS.

System	Resolved and Total Variable Features, All Patterns											
	Variables		Function Calls		Method Calls		Property Fetches		Instantiations		All	
	Resolved	Uses	Resolved	Uses	Resolved	Uses	Resolved	Uses	Resolved	Uses	Resolved	Uses
CakePHP	13	14	0	0	6	25	33	327	1	110	55	490
CodeIgniter	12	12	9	15	2	25	10	82	0	23	33	157
DoctrineORM	0	0	0	3	6	13	0	85	1	29	7	131
Drupal	1	1	4	377	2	3	11	96	1	25	19	502
Gallery	2	7	7	7	2	14	22	96	1	19	37	155
Joomla	1	3	4	12	10	37	65	648	11	210	95	940
Kohana	2	7	0	12	4	14	3	17	0	14	9	69
Magento	26	32	21	36	31	174	29	488	20	309	127	1,043
MediaWiki	5	8	1	26	11	25	22	106	9	90	48	255
Moodle	14	36	29	200	7	109	222	1,624	39	371	320	2,501
osCommerce	0	119	0	2	0	0	3	23	1	29	4	175
PEAR	1	1	2	2	6	16	4	7	2	22	15	48
phpBB	40	61	7	27	4	5	4	12	2	37	58	148
phpMyAdmin	3	30	3	33	7	12	0	13	0	18	13	112
SilverStripe	1	2	0	20	145	240	5	165	7	154	158	607
Smarty	34	40	0	12	2	22	8	12	0	22	44	108
SquirrelMail	7	24	1	24	0	0	0	3	0	0	8	51
Symfony	0	0	0	57	3	93	5	53	1	36	10	268
WordPress	0	1	10	25	1	5	34	196	1	18	46	246
ZendFramework	2	4	1	219	21	84	3	98	4	99	31	548

TABLE XII. PHP VARIABLE FEATURE ANTI-PATTERN RESULTS.

System	Results of Anti-Pattern Detection Across the Corpus											
	Variables		Function Calls		Method Calls		Property Fetches		Instantiations		All	
	Detected	Resolved	Detected	Resolved	Detected	Resolved	Detected	Resolved	Detected	Unresolved	Detected	Resolved
CakePHP	0	0	0	0	23	2	203	14	29	0	257	18
CodeIgniter	0	0	10	1	19	0	46	0	19	0	94	1
DoctrineORM	0	0	4	0	16	6	41	0	25	0	87	6
Drupal	0	0	393	2	2	2	75	7	33	1	503	12
Gallery	0	0	7	5	13	1	61	6	10	0	100	12
Joomla	0	0	11	2	29	10	464	62	140	6	661	82
Kohana	0	0	6	0	16	4	15	1	12	0	55	5
Magento	2	0	32	16	138	24	403	13	325	16	904	69
MediaWiki	1	0	24	1	20	2	59	5	78	2	182	10
Moodle	12	1	201	28	86	3	1,090	114	262	14	1,747	161
osCommerce	84	0	2	0	0	0	3	0	18	0	109	0
PEAR	0	0	0	0	16	1	2	0	19	2	37	3
phpBB	2	0	26	1	4	3	5	0	31	0	70	4
phpMyAdmin	21	2	21	2	11	7	9	0	8	0	77	11
SilverStripe	0	0	22	0	385	144	121	0	109	1	658	145
Smarty	2	0	14	0	23	0	12	8	13	0	64	8
SquirrelMail	2	0	42	0	0	0	3	0	0	0	47	0
Symfony	0	0	28	0	147	2	45	0	22	0	245	2
WordPress	0	0	27	3	4	1	189	23	15	0	236	27
ZendFramework	2	0	100	0	57	12	62	0	80	1	345	13

The results indicate that the anti-patterns are quite good at detecting scenarios that are difficult to resolve locally, e.g., where the values used to compute the identifier are provided as function parameters or as global variables. In our opinion, after manually inspecting several thousand occurrences, we believe that many of these occurrences *cannot* be statically resolved but are actually used to support dynamic features of the system, such as the ability to add user-defined metadata as new properties on specific instances of provided classes. However, while in most cases very few of the detected anti-patterns are also resolved, there are several where the number resolved is above 10 (e.g., variable functions in Magento and Moodle, or variable methods in Joomla, Magento, SilverStripe, and the Zend Framework). More research needs to be conducted to identify scenarios where both a pattern and anti-pattern match, which hopefully will allow the patterns to be further improved.

G. Threats to Validity

There are several threats to the validity of the results reported above. First, a different corpus or an individual system not in the corpus could yield results that are quite different from those reported here. To mitigate this risk we have chosen a wide variety of systems from different application domains, and have included both complete systems and commonly-used application frameworks. This can be seen in the reported results, which vary considerably across the different systems in the corpus. Second, the list of patterns is most likely not complete. While this is true, at some point an individual pattern of use in a specific application may not be widely used enough outside of this application to warrant inclusion in the patterns we have developed, and after inspecting several thousand variable feature occurrences we believe the most common patterns, used broadly in the corpus, are included (and even at this point, some are much more widely used than others). Finally, a stronger analysis, not focused on finding or exploiting patterns, may be able to resolve more occurrences to specific sets of names. While true, an important goal of this research was to identify these patterns and determine how often they are used, something that should help to increase understanding of how variable features in PHP are used in practice.

VI. Related Work

Focusing just on PHP, Eshkevari et al. [6] studied runtime type changes for variables in PHP code to see how often the ability of a variable to take on multiple types is actually use. Mulder [7] used runtime tracing to examine dynamic method and function calls in PHP to detect which functions and methods could actually be called and to generate static variants of these calls which could more easily be analyzed. In our own earlier work we focused on how static and dynamic features are used in an earlier version of the corpus studied in this paper [1] and on how the use of dynamic features has changed over time in two of the systems in the current corpus [8]. We also focused on another dynamic feature, the file inclusion mechanism, demonstrating two algorithms that were able to resolve a number of includes to either one file or a small set of files statically [9], again evaluated over an earlier version of the current corpus.

Looking beyond PHP, there are a number of studies that apply static techniques to examine how language features are used and to find patterns that can be exploited in program analysis algorithms. Hackett and Aiken [10], as part of their work on alias analysis, studied aliasing patterns in large C systems programs, identifying nine patterns that account for almost all aliasing encountered in their corpus. Ernst et al. [11] investigated use of the C preprocessor, work that was instrumental to further experiments in preprocessor-aware C code analysis and transformation, such as that performed by Garrido [12]. Liebig et al. also focused on the C preprocessor, undertaking a targeted empirical study to uncover the use of the preprocessor in encoding variations and variability [13]. Collberg et al. [14] performed an in-depth empirical analysis of Java bytecode, computing a wide variety of metrics, including object-oriented metrics (e.g., classes per package, fields per class) and instruction-level metrics (e.g., instruction frequencies, common sequences of instructions). Baxter et al. [15] looked at a combination of Java bytecode and Java source (generated from bytecode), where they focused on characterizing the distributions for a number of metrics.

Other studies have either used a combination of static and dynamic analysis techniques or have just used dynamic analysis. Knuth [16] used a combination of both static and dynamic techniques to examine real-world FORTRAN programs, gathering statistics over FORTRAN source code and using both profiling and sampling techniques to gather runtime information. Richards et al. [17] used trace analysis over runtime traces gathered with an instrumented browser to examine how the dynamic features of JavaScript are used in practice, specifically investigating whether the scenarios assumed by static analysis tools (e.g., limited use of `eval`, limited deletion of fields, uses of functions that match the provided function signatures) are accurate. In a more focused study over a larger corpus, Richards et al. [18] analyzed runtime traces to find uses of `eval`; as part of this work, the authors categorized these uses into a number of patterns. Meawad et al. [19] then used these results to create a tool, Evalorizer, that can be used to remove many uses of `eval` found in JavaScript code.

Morandat et al. [20] undertook an in-depth study of the R language, using a combination of runtime tracing and static analysis to examine a corpus of 3.9 million lines of R code so they could determine how the features in R are actually used. Furr et al. [21] profiled uses of dynamic Ruby features by running existing Ruby code using provided test cases, determining how the dynamic features of a program are used in practice. They discovered that these features are generally used in ways that are almost static, similarly to some of the variable features shown in this paper (e.g., a number of the loop patterns, like that shown in Figure 3), allowing them to replace these dynamic features with static versions that are then amenable to static type inference in a system such as DRuby [22].

VII. Conclusions

PHP's variable features are examples of dynamic language features that are widely used and that make static analysis and comprehension of PHP code challenging. In this paper we presented 23 patterns that account for a number of common usage scenarios for these features: 14 patterns using `foreach` and `for` loops; 4 using one or more assignments into a variable; and 5 that use conditional control flow, such as PHP's conditional statement and `switch` statement, in tandem with a variable that is both included in the condition and is used as part of the computation of the identifier.

We evaluated these patterns across a corpus made up of 20 open-source PHP systems consisting of $31,624$ PHP files with $3,725,904$ lines of PHP source. This evaluation showed that these patterns are quite common in PHP programs; that in some cases (such as with variable variables) many instances of these patterns are essentially static; but that, in many cases, uses of these variable features truly appear to be dynamic, requiring techniques such as runtime tracing to approximate, with any precision, the actual values that could be used. We also evaluated a small collection of three anti-patterns, identifying common scenarios that could make it hard to resolve the names used in a given variable feature (e.g., the name is based on a global variable), and showed that these patterns are generally good at identifying cases that either were not covered by our patterns or could not be resolved by them. We believe that these patterns and anti-patterns can be used by program analysis tools both to improve precision, in cases where the names can be resolved, and to indicate where names most likely cannot be resolved statically, requiring the use of dynamic analysis techniques to discover runtime behavior.

In future work we plan to extend the collection of anti-patterns to cover additional problematic cases. We would also like to better understand those cases where an anti-pattern detects a potential problem that is still resolvable to see if that could provide insight that would allow us to improve the existing patterns. Beyond this, it may also be possible to derive a set of names by using regular expression matching in cases where we have a partial string literal; this could prove especially effective for class, function, and method names, where (with the latter) inferred type information for the method target could also be used. Finally, as part of our work on dynamic analysis of PHP programs, we plan to investigate techniques that can direct tests towards uses of these features, allowing us to profile their execution and get a clearer picture of which values each occurrence can take on and where these values come from.

REFERENCES

[1] M. Hills, P. Klint, and J. J. Vinju, "An Empirical Study of PHP Feature Usage: A Static Analysis Perspective," in *Proceedings of ISSTA 2013*. ACM, 2013, pp. 325–335.

[2] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM 2009*. IEEE, 2009, pp. 168–177.

[3] P. Klint, T. van der Storm, and J. Vinju, "EASY Meta-programming with Rascal," in *Post-Proceedings of GTTSE'09*, ser. LNCS. Springer, 2011, vol. 6491, pp. 222–289.

[4] "Count Lines of Code Tool," http://cloc.sourceforge.net.

[5] M. Hills and P. Klint, "PHP AiR: Analyzing PHP Systems with Rascal," in *Proceedings of CSMR-WCRE 2014*. IEEE, 2014, pp. 454–457.

[6] L. M. Eshkevari, F. D. Santos, J. R. Cordy, and G. Antoniol, "Are PHP Applications Ready for Hack?" in *Proceedings of SANER 2015*. IEEE, 2015, pp. 63–72.

[7] C. Mulder, "Reducing Dynamic Feature Usage in PHP Code," Master's thesis, University of Amsterdam, 2013.

[8] M. Hills, "Evolution of Dynamic Feature Usage in PHP," in *Proceedings of SANER 2015*. IEEE, 2015, pp. 525–529.

[9] M. Hills, P. Klint, and J. J. Vinju, "Static, Lightweight Includes Resolution for PHP," in *Proceedings of ASE 2014*. ACM, 2014, pp. 503–514.

[10] B. Hackett and A. Aiken, "How is Aliasing Used in Systems Software?" in *Proceedings of FSE'06*. ACM, 2006, pp. 69–80.

[11] M. D. Ernst, G. J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[12] A. Garrido, "Program Refactoring in the Presence of Preprocessor Directives," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[13] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in *Proceedings of ICSE'10*. ACM, 2010, pp. 105–114.

[14] C. S. Collberg, G. Myles, and M. Stepp, "An empirical study of Java bytecode programs," *Software: Practice and Experience*, vol. 37, no. 6, pp. 581–641, 2007.

[15] G. Baxter, M. R. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. D. Tempero, "Understanding the Shape of Java Software," in *Proceedings of OOPSLA'06*. ACM, 2006, pp. 397–412.

[16] D. E. Knuth, "An Empirical Study of FORTRAN Programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[17] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *Proceedings of PLDI 2010*. ACM, 2010, pp. 1–12.

[18] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications," in *Proceedings of ECOOP 2011*, ser. LNCS, vol. 6813. Springer, 2011, pp. 52–78.

[19] F. Meawad, G. Richards, F. Morandat, and J. Vitek, "Eval Begone!: Semi-Automated Removal of Eval from JavaScript Programs," in *Proceedings of OOPSLA'12*. ACM, 2012, pp. 607–620.

[20] F. Morandat, B. Hill, L. Osvald, and J. Vitek, "Evaluating the Design of the R Language - Objects and Functions for Data Analysis," in *Proceedings of ECOOP'12*, ser. LNCS, vol. 7313. Springer, 2012, pp. 104–131.

[21] M. Furr, J. hoon (David) An, and J. S. Foster, "Profile-Guided Static Typing for Dynamic Scripting Languages," in *Proceedings of OOPSLA 2009*. ACM, 2009, pp. 283–300.

[22] M. Furr, J. An, J. S. Foster, and M. W. Hicks, "Static Type Inference for Ruby," in *Proceedings of SAC'09*. ACM, 2009, pp. 1859–1866.