An Empirical Study of PHP Feature Usage

A Static Analysis Perspective

Mark Hills[1], Paul Klint[1,2], and Jurgen Vinju[1,2]
[1]Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2]INRIA Lille Nord Europe, Lille, France
Mark.Hills,Paul.Klint,Jurgen.Vinju@cwi.nl

ABSTRACT

PHP is one of the most popular languages for server-side application development. The language is highly dynamic, providing programmers with a large amount of flexibility. However, these dynamic features also have a cost, making it difficult to apply traditional static analysis techniques used in standard code analysis and transformation tools. As part of our work on creating analysis tools for PHP, we have conducted a study over a significant corpus of open-source PHP systems, looking at the sizes of actual PHP programs, which features of PHP are actually used, how often dynamic features appear, and how distributed these features are across the files that make up a PHP website. We have also looked at whether uses of these dynamic features are truly dynamic or are, in some cases, statically understandable, allowing us to identify specific patterns of use which can then be taken into account to build more precise tools. We believe this work will be of interest to creators of analysis tools for PHP, and that the methodology we present can be leveraged for other dynamic languages with similar features.

1. INTRODUCTION

PHP [2], invented by Rasmus Lerdorf in 1994, is an imperative, object-oriented language focused on server-side application development. It is now one of the most popular languages, as of January 2013 ranking 6th on the TIOBE programming community index [5], used by 78.8 percent of all websites whose server-side language can be determined [4], and ranking as the 6th most popular language on GitHub [3]. PHP is dynamically typed, with a single-inheritance class model and a number of standard built-in types (e.g., strings, integers, floats). Type correctness is judged based on *duck typing*, allowing values to be used whenever they can behave like values of the expected type. For instance, adding the strings `"3"` and `"4"` yields the number 7, while concatenating the numbers 3 and 4 yields the string `"34"`. PHP also includes a number of object-oriented features, such as interfaces, exceptions, and traits, as well as dynamic features common to other scripting languages, including an `eval` expression to run dynamically-built code and special methods that handle uses of undefined object fields and methods.

The flexible, dynamic nature of PHP may sometimes yield unexpected results or make programs harder to understand: the files that are included in another file are computed at runtime, making it difficult to know, before execution, the text of the program that will actually run; variable constructs provide reflective access to variables, classes, functions, methods, and properties through strings, which may be defined dynamically, or may even be based on user input; magic methods make it hard to reason about which fields and methods a class actually supports; `eval` can run arbitrary, potentially dangerous code, and can create new definitions, including entire new classes; and the behavior of built-in operations can be puzzling, returning unexpected results (e.g., `"hello"+"world"` is equal to 0). While these features provide strong motivation for building program analysis tools to aid in program understanding, testing, security vulnerability detection, refactoring, and debugging, they also make it hard for these tools to efficiently provide correct, precise results.

To understand the impact of these features on analysis tool development, we set out to answer the following questions. How large are real PHP programs? How often are the various features of the PHP language used in these programs? With the goal of supporting a core of the language for experimentation with new analysis algorithms and tools, which features would need to be defined precisely to faithfully capture the core of PHP as it is used in practice, and which little-used features could be modeled with less precision? Where and how often are some of the harder to analyze language features (such as those mentioned above, like dynamic includes) used in existing PHP code? Are these features spread evenly through the code, or do they tend to cluster in specific files? And, finally, are these uses truly dynamic, or is it possible to capture patterns that can be leveraged in analysis tools?

To answer these questions, we assembled a large corpus of open-source PHP systems, described further in Section 3. We then analyzed this corpus using the Rascal [14] metaprogramming language (Section 4). Sections 5 and 6 contain our main results with answers to the questions posed above (the first three, on general feature usage, in Section 5, the last three, specifically on the dynamic features of PHP, in Section 6). Section 7 then presents final thoughts and concludes. But first, in Section 2 we discuss related work focused on empirical studies of feature usage in real programs.

System	Version	PHP	Release Date	File Count	SLOC	Description
CakePHP	2.2.0-0	5.2.8	2012-07-02	640	137,900	Application Framework
CodeIgniter	2.1.2	5.1.6	2012-06-29	147	24,386	Application Framework
Doctrine ORM	2.2.2	5.3.0	2012-04-13	501	40,870	Object-Relational Mapping
Drupal	7.14	5.2.4	2012-05-02	268	88,392	CMS
Gallery	3.0.4	5.2.3	2012-06-12	505	38,123	Photo Management
Joomla	2.5.4	5.2.4	2012-05-02	1,481	152,218	CMS
Kohana	3.2	5.3.0	2011-07-25	432	27,230	Application Framework
MediaWiki	1.19.1	5.2.3	2012-06-13	1,480	846,621	Wiki
Moodle	2.3	5.3.2	2012-06-25	5,367	729,337	Online Learning
osCommerce	2.3.1	4.0.0	2010-11-15	529	44,952	Online Retail
PEAR	1.9.4	4.4.0	2011-07-07	74	31,257	Component Framework
phpBB	3	4.3.3	2012-01-12	269	148,276	Bulletin Board
phpMyAdmin	3.5.0	5.2.0	2012-04-07	341	116,630	Database Administration
SilverStripe	2.4.7	5.2.0	2012-04-05	514	108,220	CMS
Smarty	3.1.11	5.2.0	2012-06-30	126	15,468	Template Engine
Squirrel Mail	1.4.22	4.1.0	2011-07-12	276	36,082	Webmail
Symfony	2.0.12	5.3.2	2012-03-19	2,137	120,317	Application Framework
WordPress	3.4	5.2.4	2012-06-13	387	110,190	Blog
The Zend Framework	1.11.12	5.2.4	2012-06-22	4,342	553,750	Application Framework

The PHP versions listed above in column PHP are the minimum required versions. The File Count includes files with a .php or an .inc extension. In total there are 19 systems consisting of 19,816 files with 3,370,219 total lines of source.

Table 1: The PHP Corpus.

2. RELATED WORK

We are not aware of any empirical studies of the PHP language similar to our own. However, there are a number of studies focused on applying static techniques to other languages, examining how language features are used or looking for patterns that can be exploited in analysis. Hackett and Aiken [12], as part of their work on alias analysis, studied aliasing patterns in large C system programs, identifying nine patterns that account for almost all aliasing encountered in their corpus. Ernst et al. [8] investigated similar questions to those we have for PHP, but for the C preprocessor. This result was instrumental in the development of further experiments in preprocessor-aware C code analysis and transformation, such as that by Garrido [11]. 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 [16]. Collberg et al. [7] 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. [6] 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 used a combination of static and dynamic analysis, or have instead focused just on dynamic analysis—for instance, by gathering execution traces which are then used to provide information on runtime behavior. Knuth [15] 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. [20] used trace analysis 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, use of functions that matches the provided function signatures) are accurate. In a more focused study over a larger corpus, Richards et al. [19] 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. [17] then used these results to create a tool, Evalorizer, that can be used to remove many uses of `eval` found in JavaScript code. Sridharan et al. [21] show that scaling analysis tools to handle real-world code is challenging, but that leveraging idiomatic uses of dynamic JavaScript features can greatly improve the efficiency and precision of static analyses, something we are attempting to do as well in identifying usage patterns of dynamic features in PHP code.

Morandat et al. [18] undertook an in-depth study of the R language. As part of this work, they used a combination of runtime tracing and static analysis to examine how the unlikely combination of features found in R are actually used, examining a corpus of 3.9 million lines of R code pulled from multiple repositories. Furr et al. [9] used profiling of dynamic Ruby features, in conjunction with provided test cases, to determine 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, allowing them to replace these dynamic features with static versions that are then amenable to static type inference in a system such as DRuby [10]. We have not used this approach, but, based on our results, believe it could be applied to PHP as well.

3. CORPUS

We assembled a corpus of 19 large open-source PHP systems, basing our choice on popularity rankings provided by Ohloh[1], a site that tracks open-source projects. The chosen systems are shown in Table 1. Systems were generally selected just 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. We used popularity, instead of actual number of downloads or installed sites, since we have

[1]`http://www.ohloh.net/tags/php`

no way to accurately compute these figures. In total, the corpus consists of 19,816 PHP source files with 3,370,219 lines of PHP source (counted using the cloc [1] tool).

When building the corpus, we chose specifically to focus on larger, more widely used systems for two reasons. First, such large systems are more likely to benefit from static analysis and the code inspection and transformation tools such analysis enables. Second, larger systems should provide a greater mix of language features and usage scenarios. However, this does present two related threats to validity: a different corpus could yield very different results, and our selection of larger, more mainstream projects may mean we are seeing more carefully written code than is actually typical of many real-world PHP programs. For the first, we believe that choosing systems across a variety of different application areas helps to ensure the corpus is representative of actual PHP use. For the second, we believe this is a natural consequence of selecting popular, widely-used systems for the corpus, but even with that said, there are still significant differences in how features are used in the different systems, with some taking much more advantage of newer code structuring features (e.g., PHP5 classes and namespaces) than others.

4. RESEARCH METHOD

Rascal [14], a meta-programming language for source code analysis and transformation, uses a familiar Java-like syntax and is based on immutable data (trees, relations, sets), term rewriting, and relational calculus primitives. All analysis performed in this paper was performed using Rascal code, ensuring that the results are reproducible and checkable. We also used Rascal to generate the LaTeX for the tables and `pgfplots` figures in this paper, using the Rascal string template facility to produce the LaTeX code found in the LaTeX source of the paper. All code is available online at `https://github.com/cwi-swat/php-analysis`.

We are parsing PHP scripts using our fork[2] of an open-source PHP parser[3], which itself is 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. Of the 19,816 files in the corpus analyzed in this paper, 11 fail to parse. 10 of these files are actually uninstantiated templates which would need to be instantiated to be valid PHP, while the other, in the Zend Framework, is a test file designed to trigger a parse error. Reuse of an existing PHP parser provides a way for us to stay compatible with changes to PHP as it evolves.

A number of different functions extract data from the source code. For each file we measure the lines of code and the language features—expressions, statements, and declarations—used in the file, based on our count of the AST nodes used to represent these features. Information on how dynamic features are used is computed by extracting occurrences of specific patterns from the data, such as of `include` expressions that do not use static paths, definitions of methods used to handle reads and writes to missing fields, and calls to standard functions used to support dynamic language features (e.g., `create_function`). Some information is instead based on directly inspecting the underlying code. This information is stored in CSV files, which can then be read in as Rascal relations and used as part of other computations or to generate the tables and figures in the paper.

[2]`https://github.com/cwi-swat/PHP-Parser`
[3]`https://github.com/nikic/PHP-Parser/`

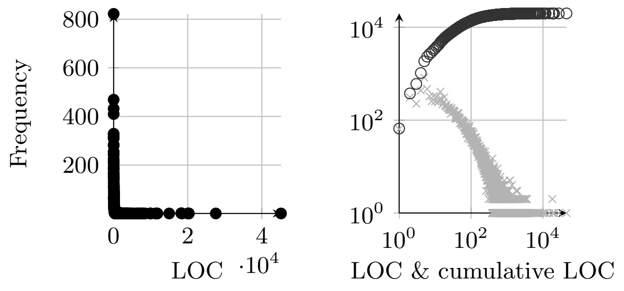

Figure 1: PHP File Sizes, Linear and Log Scales.

Distribution graphs are directly generated from the metrics data, without any statistical analysis in between. We use log axes when the plots are indistinguishable otherwise. We normalize for the size of a PHP file by considering the ratio between a single feature and the total number of feature usages detected in a PHP file (see Section 5.2). Computation of feature coverage is a combinatorial optimization problem, which we approximate as described in Section 5.3.

5. PHP IN GENERAL

In this section we provide an overview of the usage of all PHP language features in the assembled PHP corpus, answering the first three questions we posed in Section 1. First, taking the view that a system is made up of a number of individual programs—each a collection of files—that run in response to user requests, we look at the sizes of the individual files that make up the programs in the corpus (dynamic includes, described in Section 6, make it impossible to statically look at each program as a whole). Next, we look at how the various features of the PHP language are distributed across these program files. Finally, with the goal of defining a core PHP for experimentation, we look at which features of PHP would need to be defined precisely to faithfully capture the core of the language as it is used in practice, meaning this core should fully support a large number of real programs.

5.1 PHP File Size

First we must understand how PHP file sizes are distributed. Naturally, larger files contain more individual uses of features; we would also expect to see a wider variety of features used as files increase in size. This means feature usage expectancy is likely to be dominated by the file size.

The histogram in Figure 1 depicts the distribution of file sizes in the corpus, left on a linear scale, and right on a logarithmic scale (bottom, gray x's) with the cumulative distribution superimposed on it (top, black o's). The distribution of sizes over files has a very high start (there are many very short files) and a very long tail (there are some very large files). Only when we plot the distribution logarithmically on both axes can we really see its shape. The lower limit of the distribution drops off faster than negative exponential (it curves down on the log scale).

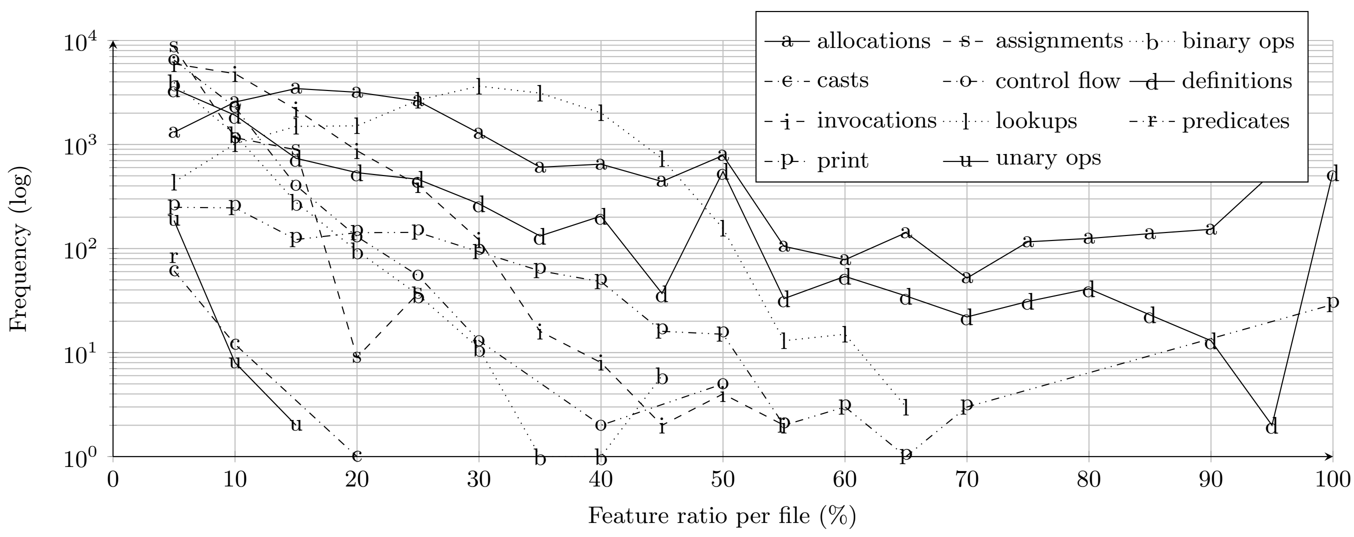

Figure 2: What features should one expect to find in a given PHP file? This histogram shows, for each feature group, how many times it covers a certain percentage of the total number of features per file. Lines between dots are guidelines for the eye only.

allocations	array, <u>clone</u>, new, nextArrayElem, scalar
assignments	*BitAnd*, *BitOr*, *BitXor*, Concat, *Div*, *LShift*, <u>Minus</u>, *Mod*, *Mul*, Plus, *RShift*, assign, listAssign, refAssign, unset
binary ops	<u>BitAnd</u>, <u>BitOr</u>, *BitXor*, BoolAnd, BoolOr, Concat, Div, Equal, Geq, Gt, Identical, *LShift*, Leq, LogAnd, LogOr, *LogXor*, Lt, Minus, Mod, Mul, NotEqual, NotId, Plus, *RShift*
casts	toArray, toBool, *toFloat*, toInt, <u>toObject</u>, toString, **toUnset**
control flow	break, continue, *declare*, <u>do</u>, exit, expStmt, for, foreach, **goto**, **haltCompiler**, if, **label**, return, suppress, switch, ternary, throw, tryCatch, while
definitions	classConstDef, classDef, *closure*, **const**, functionDef, global, include, interfaceDef, methodDef, namespace, propertyDef, static, **traitDef**, use
invocations	call, *eval*, methodCall, *shellExec*, staticCall
lookups	fetchClassConst, fetchConst, fetchStaticProperty, propertyFetch, **traitUse**, var
predicates	empty, instanceOf, isSet
print	echo, inlineHTML, <u>print</u>
unary ops	*BitNot*, BoolNot, <u>PostDec</u>, PostInc, *PreDec*, PreInc, UnaryMinus, *UnaryPlus*

Features in **bold** are not used in the corpus. Features in *italics* are not used in 90% of the corpus files. Features that are <u>underlined</u> are not used in 80% of the corpus files.

Table 2: Logical Groups of PHP Features.

It is unnecessary to characterize the type of distribution our analysis yields (which would be challenging to validate). Nevertheless, as can be estimated from the cumulative distribution graph, the area under 1303 LOC covers 98% of the corpus. Although there are 397 files with larger sizes, they do not contribute significantly to the size of the corpus.

5.2 PHP Feature Distribution

We have grouped the PHP language features into 11 categories (Table 2). Syntax analysis reveals that 102 different PHP language features are used in the corpus, out of 109 total features (unused features are shown in Table 2 **in bold**).

If we were to plot the feature distributions over files in our corpus, all plots would neatly follow the shape of the above file size distribution. Instead of doing this, we normalize for the file size by computing for every feature, for every file, the ratio between this feature and the total number of feature usages in the file. Figure 2 plots a histogram of these ratios for each of the aforementioned groups of features. You can read from this graph what features to expect when you open an arbitrary file in the corpus.

The graph shows how the bulk of our corpus consists of files that have high variety in feature usage. Casts, unary operators and builtin predicates are almost never used, and never in large quantities in the same file. When we move to the right —as variety diminishes— we see that lookups and allocations rise in exchange for invocations, control flow, definitions, prints, and binary operations. Notably, the distributions for lookups and allocations are different: they have strong upward curves and would look bell shaped when printed on a linear axis. For these feature groups we can predict what the most likely percentage is, which is where they reach the maximum coverage. For lookups this is 30% (±10%), for allocations 15% (±10%).

The distribution for definitions (functions, classes, methods) is interesting. It drops off rapidly (exponentially) from more than half of the files that contain practically no definitions, to 35 files that consist of around 45% definitions. Then it spikes again at 50% to 551. A quick inspection shows that these 551 files make heavy use of PHP's object-oriented definition features, defining interfaces, classes, and methods, while the other files contain more "procedural" code (i.e., regular statements and expressions from the top-level of the script or from method and function bodies). There are also hundreds of files which consist solely of definitions (100%). These are abstract classes and interfaces.

All the way on the right, the most uniform files are represented. For example, there are 100 files where 90% of the

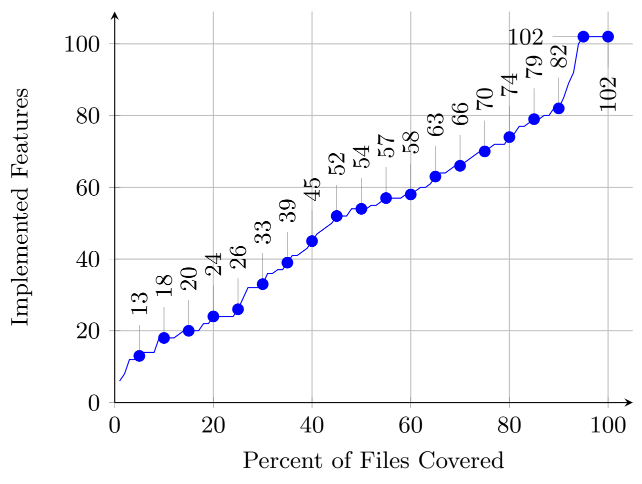

Figure 3: Features Needed for Percent File Coverage. Numbers of features are shown for each 5% increment in coverage. There are 109 features total.

features are data allocations. Typically those are files which contain large arrays of constants, for string translations or character encoding. We also see a few dozen files which consist of prints only. These are the HTML template files. It is surprising to see such a low number of these in a PHP corpus, but only if we forget that the corpus is made up of general applications and reusable frameworks that, even when used in scripts that return HTML, do not necessarily directly generate HTML themselves.

Now we have an overview that shows that:

- Practically all files are below 1300 LOC;
- Seven out of 109 features are never used, and predicates, casts and unary operations are used only sporadically;
- There is no immediate "core" of PHP features that cover the larger part of many files, but allocation, lookup and printing, wrapped in definitions such as variables, functions, and methods, can be expected to be the most prominent groups of features in any given file.
- Four types of PHP files are distinguishable based on the balance of features used: object-oriented definitions, HTML templates, data definitions, and procedural, the latter mainly consisting of statements and expressions from the top-level of the script or in the bodies of functions and methods. Most files are procedural.

5.3 PHP Feature Coverage

To build an effective core PHP for experimenting with new analysis algorithms and tools, it is important to know which language features we need to include to faithfully capture how PHP is used in practice. While the information in Section 5.2 provides guidance about what we can expect to find in any given PHP file, it does not tell us which features we need to implement to completely *cover* a file, i.e., to ensure that all features used in the file are implemented. To provide guidance about the feature set to actually implement, we ask the following question: what language features must be supported to cover all features used in a given percentage of the files in the corpus? This should allow us to capture the core of the language, while ensuring experiments using this core can actually be used on real code.

System	80% set	90% set	System	80% set	90% set
CakePHP	95.3%	98.3%	MediaWiki	86.1%	94.6%
osCommerce	95.1%	96.4%	SilverStripe	85.4%	91.1%
ZendFramework	93.2%	97.3%	phpMyAdmin	85.3%	90.3%
Kohana	92.1%	96.5%	WordPress	82.4%	95.1%
Symfony	91.1%	94.9%	Gallery	81.0%	96.6%
Joomla	91.0%	97.0%	PEAR	75.7%	90.5%
SquirrelMail	90.9%	95.7%	phpBB	72.1%	85.1%
DoctrineORM	89.2%	96.6%	Smarty	66.7%	86.5%
Moodle	87.6%	96.9%	Drupal	57.1%	93.7%
CodeIgniter	87.1%	91.8%			

Table 3: Percent of System Covered by Feature Sets.

Selecting this smallest set of features is non-trivial. With 109 features available, the smallest set of features that covers x% of the corpus is one of the 2^{109} combinations of features, and need not be unique (there could be multiple smallest sets of the same size). Ergo, a brute force solution to find this is infeasible. Similarly to standard combinatorial problems such as Set-Cover and Maximum-Coverage, we instead approximate the solution using a greedy algorithm. Our polynomial time algorithm starts with all the features that need to be present to reach a given percentage (e.g., a feature that appears in 85% of all files needs to be included in any feature set that covers at least 15% of the files) and then greedily adds features based on the feature *popularity*, i.e., how many files it appears in. This is continued until the target percentage is reached.

The results of this analysis, for coverage of 1% to 100% of the corpus, are shown in Figure 3. The numbers given in the graph are the number of features sufficient to cover that percentage of files, with each 5% increment labeled with this number of features. For example, to cover 50% of the files the language core must support 54 PHP language features, while 74 features are enough to cover 80% of the corpus.

Every 10% of added coverage adds roughly 10 more language features. The nearly linear increase of file coverage for every feature added confirms the variety in the distribution of feature usage over files that we observed on the left-hand side of Figure 2. Although in theory our analysis only shows which features are sufficient, not which are necessary, it would be surprising if the current graph does not approximate the optimal solution rather closely. Smaller solutions would entail the existence of small groups of features that, in combination, cover large parts ($> 10\%$) of the corpus. While this is possible, Figure 2 indicates that the existence of any such group is unlikely, let alone several of them.

The bottom line of this analysis is that, to create a core PHP for experimentation that still handles all the features in 80% of the corpus files, 74 features should be defined, while 82 features are needed to cover 90% of the corpus files. These features are shown in Table 2 by *italicizing* features not needed for the 90% case and underlining features also not needed for the 80% case. Assuming a core PHP is defined

```
$deps = "{$wgStyleDirectory}/{$skinName}.deps.php";
if (file_exists($deps)) {
        include_once($deps);
}
require_once("{$wgStyleDirectory}/{$skinName}.php");
```

Figure 4: Dynamic Includes.

with these feature sets, Table 3 then shows how much of each of the corpus systems would be covered. For instance, a core PHP defined using the 80% feature set would actually cover 95.3% of the files in CakePHP. One threat to validity with these figures is that include expressions could make the distribution of features in a program much different than the distributions we see in individual files. However, as discussed in Section 6, we have (except in isolated cases) not seen this happen with the dynamic features we looked at, even in cases where many of the dynamic includes can be resolved.

6. DYNAMIC PHP LANGUAGE FEATURES

The PHP language includes a number of dynamic features that can be challenging to model in static analysis tools. Below, we look at six of these features: dynamic file includes; variable constructs, which allow variables to be used in place of identifiers to name entities such as variables (variable variables) and functions (variable functions); overloading, which uses so-called magic methods to dynamically handle accesses of undefined or non-visible methods and properties; the `eval` expression; variadic functions; and `call_user_func`, a function used to call other functions, taking the function name and arguments as parameters. For each feature, we focus on answering the final three questions in Section 1. First, we look at where, and how often, these features are used in PHP programs. Next we look at how uses of these features are distributed: are they clustered together, or spread evenly through the files in which they appear? Finally, with the first two features and with `eval` we look at how dynamic the features are in practice, looking for usage patterns and, with dynamic includes, briefly discussing the results of an analysis that resolves many apparently dynamic cases.

6.1 Dynamic Includes

In PHP, a script includes another file using an `include` expression (including the variants `include_once`, `require`, and `require_once`). The name of the file to include can be provided as a string literal, but can also be dynamic, given using an arbitrary expression computed at runtime. Because of this, it may not be possible for static analysis tools to know the PHP source code for the program that will actually be executed. Two examples, from `includes/Skin.php` in MediaWiki 1.19.1, are shown in Figure 4: `$deps`, a string based on a combination of a global variable, a local variable, and a string literal, names the file included on line 3; a second file, identified by the same path as the first except for a different string literal, is included on line 5.

Table 4 provides a high-level overview of the incidence of dynamic includes in the corpus. The total number of include expressions in the system is shown in column *Total*, with anywhere from just 38 includes in Smarty to 12,829 in the Zend Framework. The next column, *Dynamic*, then restricts this number to just those includes with dynamic paths, defined here as any path not given solely by a string literal.

System	Includes			Files	Gini
	Total	Dynamic	Resolved		
CakePHP	124	120	91	640(19)	0.28
CodeIgniter	69	69	28	147(20)	0.44
DoctrineORM	56	54	36	501(14)	0.19
Drupal	172	171	130	268(16)	0.42
Gallery	44	39	25	505(10)	0.26
Joomla	354	352	200	1,481(122)	0.17
Kohana	52	48	4	432(18)	0.55
MediaWiki	554	493	425	1,480(38)	0.34
Moodle	7,744	4,291	3,350	5,367(504)	0.39
osCommerce	683	539	497	529(22)	0.28
PEAR	211	11	0	74(9)	0.14
phpBB	404	404	313	269(51)	0.34
phpMyAdmin	819	52	15	341(27)	0.23
SilverStripe	373	56	27	514(10)	0.34
Smarty	38	36	25	126(7)	0.29
SquirrelMail	426	422	406	276(13)	0.14
Symfony	96	95	41	2,137(40)	0.22
WordPress	589	360	332	387(17)	0.32
ZendFramework	12,829	350	285	4,342(42)	0.29

Table 4: Usage of Dynamic Includes.

As part of our ongoing work on PHP analysis, we are investigating techniques to resolve dynamic includes statically. We do this using a combination of techniques, including constant propagation, algebraic simplification (mainly for string concatenation), pattern matching over paths, and function simulation. Column *Resolved* shows the result of applying our current include resolution analysis to the dynamic includes in the corpus. While in some cases this does very little (0 resolved in PEAR, only 4 of 48 resolved in Kohana), in other cases it is quite effective, for instance resolving 332 of the 360 dynamic includes in WordPress, and 3350 of the 4291 dynamic includes in Moodle. Overall, more than 78% of the dynamic includes in the corpus are actually static.

The final two columns provide information about the resulting systems after resolving dynamic includes. Column *Files* shows the total number of files in the system (initially shown in Table 1, repeated here for convenience) along with the number of files that still contain unresolved dynamic includes, given in parentheses. Column *Gini* shows how occurences of the dynamic includes are distributed across the files which contain at least one occurrence. This is shown in terms of the Gini coefficient (from here on, just "Gini"). The Gini, ranging from 0.0 to 1.0, provides a measure for inequality—0.0 means that all files have the same number of occurrences, while 1.0 means that one file holds all the occurrences.

When we measure the Gini, here and with the other dynamic features, we only include those files with at least one occurrence of the feature we are discussing. This is done to lower noise in the computed Gini—in most cases, many of the files have no occurrences of the feature being examined. If we included these files in the Gini computation, this would cause the Gini value to be very high (i.e., very unequal) in almost all cases. Focusing on just files with occurrences, we can see more clearly how the occurrences are distributed through these files. It's also worth noting that the Gini grows very

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	w/Inc	Uses	Gini
CakePHP	640	7	20	0	0	15	25	55	377	39	95	91	92	534	0.63
CodeIgniter	147	4	20	5	6	11	17	22	59	9	14	35	36	116	0.44
DoctrineORM	501	0	0	7	15	8	8	5	60	11	21	28	29	108	0.63
Drupal	268	1	1	33	372	2	3	20	91	13	25	50	65	492	0.73
Gallery	505	3	7	3	7	6	14	25	94	13	19	46	48	153	0.52
Joomla	1,481	1	2	6	9	10	11	57	239	45	155	101	113	418	0.61
Kohana	432	3	7	3	8	4	11	6	14	11	12	24	24	56	0.44
MediaWiki	1,480	6	11	3	3	11	12	45	95	72	90	125	282	213	0.30
Moodle	5,367	19	39	68	203	61	88	248	1,276	170	387	472	1,410	2,020	0.59
osCommerce	529	21	89	1	2	0	0	4	7	15	19	38	60	117	0.45
PEAR	74	1	1	1	2	7	16	2	7	16	22	23	23	48	0.38
phpBB	269	18	82	8	36	5	6	5	14	19	27	47	85	165	0.49
phpMyAdmin	341	13	112	12	34	4	6	4	8	8	8	36	36	168	0.65
SilverStripe	514	2	3	2	2	44	102	47	152	55	173	108	116	432	0.59
Smarty	126	10	40	6	12	6	19	5	12	11	21	31	32	104	0.43
SquirrelMail	276	5	24	13	24	0	0	2	3	0	0	18	47	51	0.47
Symfony	2,137	0	0	21	37	20	22	13	98	38	57	89	90	223	0.53
WordPress	387	14	37	8	33	3	4	40	119	13	108	70	115	301	0.60
ZendFramework	4,342	4	7	7	10	93	204	120	473	151	249	320	334	947	0.50

Table 5: Usage of Variable Features.

```
 1 foreach (array('columns', 'indexes') as $var) {
 2   if (is_array(${$var})) {
 3     ${$var} = implode($join[$var], array_filter(${$var}));
 4   }
 5 }
 6
 7 foreach (array_keys(Router::getNamedExpressions()) as $var){
 8   unset(${$var});
 9 }
10
11 foreach (array('_ci_library_paths', '_ci_model_paths',
12                '_ci_helper_paths') as $var) {
13   if (($key = array_search($path, $this->{$var})) !== FALSE){
14     unset($this->{$var}[$key]);
15   }
16 }
```

Figure 5: Variable Variables and Properties.

slowly, with even figures well below 0.5 representing high inequality. For instance, if 5 files each have 1 occurence, and 1 has 2, this yields a Gini of 0.12; if 5 files have 1 and 1 has 5, this is 0.33; if 5 files have 1 and 1 has 10, this is 0.5; and if 5 have 2 and 1 has 20, this gives a Gini of 0.51. Looking at Table 4, systems such as Kohana have a high Gini, meaning that the distribution of dynamic includes is quite uneven, while a system like SquirrelMail has a low Gini, with the dynamic includes distributed much more evenly. The details bear this out: in Kohana, 15 files have 1 dynamic include, 1 has 2, 1 has 3, and 1 has 24, while in SquirrelMail 10 files have 1 dynamic include and 3 have 2.

6.2 Variable Constructs

PHP includes a number of *variable* constructs, which allow a variable that can be used as a string to be used in place of an identifier. Commonly used variants include variable variables; variable function and method calls, where the function or method name is a variable; variable class instantiations, where the class name given in a `new` expression is a variable; and variable properties, where the property name is a variable. In conjunction with reflective functions in the PHP library, these constructs allow for reflection in PHP code, and also provide a method to apply the same code to a number of different named variables or object properties. The variable features in PHP pose an obvious challenge for static analysis tools. Reads and writes through variable variables could be reads and writes to any variable in scope, including global variables; invocations of variable methods could be calls to any method supported by the target object; and so forth. Certain scenarios, like taking a reference through a variable variable, could introduce may-aliases between any of the variables in scope.

Figure 5 presents several examples of PHP variable constructs in use. In the first snippet (lines 1–5), variable variables are used as a code saving device, allowing the same logic to be applied to two variables, `$columns` and `$indexes`, by applying it instead to `${$var}`, with `$var` assigned the name of each variable in turn as part of the foreach loop. In the second snippet (lines 7–9), variable variables are again used, but here over a list of names returned by a call to method `Router::getNamedExpressions()`. In the third snippet (lines 11–16), variable properties are used to apply the same code over the list of properties given in the array, specifically in the calls to `array_search` and `unset`.

6.2.1 Findings

Table 5 provides details on the use of the variable constructs mentioned above. The first two columns show the name of the system and the number of source files. Usage information on variable variables, function calls, method calls, property fetches, and instantiations is then shown with two columns each, showing the number of files that contain one or more uses of the feature as well as the total number of uses of the feature across all files (e.g., CakePHP has 20 variable variable uses distributed over 7 files). The final four columns show usage details for all variable constructs in PHP, including those shown in the columns to the left plus several not listed here. The first and third of these columns again show the number of files containing a variable feature and the number of uses across all files, as was given for the individual features to the left. The second of the four columns then shows the number of files that include at least one variable feature if we take file inclusion into account. For instance, in MediaWiki

System	Variable-Variable Uses		
	Derivable Names	Other	Total
CakePHP	19	1	20
CodeIgniter	16	4	20
Drupal	1	0	1
Gallery	2	5	7
Joomla	0	2	2
Kohana	5	2	7
MediaWiki	5	6	11
Moodle	29	10	39
osCommerce	0	89	89
PEAR	1	0	1
phpBB	62	20	82
phpMyAdmin	86	26	112
SilverStripe	1	2	3
Smarty	38	2	40
SquirrelMail	10	14	24
WordPress	28	9	37
ZendFramework	5	2	7

Across all systems, 61.35% of the uses have derivable names. In those systems that use PHP5, 76.8% of the uses have derivable names.

Table 6: Derivability of Variable-Variable Names.

125 files contain at least one variable feature; this grows to 282 if we also count files that do not contain variable features, but (transitively) include files that do. One threat to validity with this number, here and in the remainder of this section, is that dynamic includes could indirectly bring large numbers of dynamic features into even those files that currently show few or no uses of them. Our work on statically resolving these includes, which we have used in calculating these numbers, helps to lessen this threat. The final column shows the Gini coefficient, computed across the distribution of variable features in those files that contain at least one.

From the data, it is clear that uses of PHP's variable features are not rare, but they also are not a generally used feature that is regularly encountered in program files. This seems to be especially true as the system gets larger: in the five largest systems (by file count), fewer than 10% of the files directly contain variable features. Uses of the features also tend to cluster: 11 of the systems have a Gini of at least 0.5, while 17 have a Gini of at least 0.4, and the lowest is still a fairly high 0.30.

6.2.2 *Variable Variable Usage Patterns*

To get a better idea of how variable variables are used in PHP programs, we looked at all 502 occurrences shown in Table 5. Our goal was to determine whether, at each usage site, the set of variables possibly referenced through the variable variable could be computed statically. We consider this set to be statically computable when the variable names it contains are all specified (i.e., bound as the value of the variable used as a variable variable) in one of the following ways: as string literals in an array used in a foreach statement; as a string literal in an equality comparison in a conditional; as the case condition in a case in a switch/case statement; or as a string literal (including with the above cases) built with a constrained set of string operations (e.g., a literal appended with a number between 1 and 5, or a substring of a literal). An example of where the set of names can be computed statically was shown in Figure 5 in the first snippet: at each use of the variable variable `$var`, it must refer to either variable `$columns` or variable `$indexes`, as these are given as literals in the array which provides the values for `$var`, and `$var` is not otherwise defined.

System	Files		Magic Methods						GC
	MM	WI	S	G	I	U	C	SC	
CakePHP	18	18	5	12	7	0	10	0	0.28
CodeIgniter	4	5	1	5	0	0	1	0	0.32
DoctrineORM	4	4	1	1	0	0	3	0	0.15
Drupal	2	13	0	1	0	0	1	0	0.00
Gallery	26	26	4	15	2	1	15	0	0.24
Joomla	10	10	2	7	1	1	4	0	0.26
Kohana	2	2	2	2	2	2	0	0	0.00
MediaWiki	14	14	2	3	0	0	14	0	0.21
Moodle	61	1,030	27	41	9	3	31	0	0.26
osCommerce	0	0	0	0	0	0	0	0	N/A
PEAR	1	1	0	0	0	0	1	0	N/A
phpBB	0	0	0	0	0	0	0	0	N/A
phpMyAdmin	2	2	1	1	0	0	1	0	0.17
SilverStripe	9	9	3	5	1	0	9	0	0.37
Smarty	7	8	5	6	0	0	1	0	0.12
SquirrelMail	0	0	0	0	0	0	0	0	N/A
Symfony	6	6	2	1	0	0	4	0	0.12
WordPress	4	9	2	3	2	0	0	0	0.25
ZendFramework	133	134	55	75	37	24	81	0	0.32

Table 7: Usage of Overloading (Magic Methods).

The summary of our findings is shown in Table 6. The system name and total number of variable variable uses are given in the first and fourth columns, respectively, and are identical to those shown in Table 5. The second and third columns show the number of cases where the set of variable names is statically computable, or, respectively, is not. In total, we found that, in roughly 61% of the variable variable uses in the corpus, and in roughly 76% of the uses in systems that require PHP5, the actual variables referenced using variable variables are statically determinable. Many of these uses are to reduce code repetition. By contrast, many of the cases where the names cannot be statically determined are for truly reflective operations, such as clearing the values out of all global variables representing form POST data (related to the deprecated PHP `register_globals` feature) or checking to see if all the variables in a list of variable names are set.

6.3 Overloading

In PHP, *overloading* allows specially-named methods, called *magic methods*, to support uses of properties and methods that have not been defined or are not visible (e.g., are declared `private`). Magic methods used for overloading include `__get`, to read a property; `__set`, to write a property; `__isset`, to see if a property has been set; `__unset`, to unset a property; `__call`, to invoke an instance method; and `__callStatic`, to invoke a static method.

Table 7 provides an overview of the occurrence of overloading definitions in the corpus. Column *MM* shows the number of files that contain one or more magic methods; column *WI* then extends this to consider includes, as was done above for variable features (and including the same caveats). Columns *S*, *G*, *I*, *U*, *C*, and *SC* then show the number of instances of the `__set`, `__get`, `__isset`, `__unset`, `__call`, and `__callStatic` methods, respectively. Finally, column *Gini* shows the Gini coefficient calculated over the distribution of magic methods in files containing at least one, providing a measure of how evenly these are distributed in implementing

System	Files			Total Uses	Gini
	Total	EV	WI		
CakePHP	640	3	3	5/1	0.33
CodeIgniter	147	2	2	3/0	0.17
DoctrineORM	501	0	0	0/0	N/A
Drupal	268	1	1	1/0	N/A
Gallery	505	5	7	1/4	0.00
Joomla	1,481	6	7	7/1	0.21
Kohana	432	3	3	1/2	0.00
MediaWiki	1,480	5	5	4/1	0.00
Moodle	5,367	39	1,077	34/30	0.30
osCommerce	529	4	4	4/0	0.00
PEAR	74	7	7	2/6	0.11
phpBB	269	7	24	14/1	0.40
phpMyAdmin	341	8	8	9/6	0.38
SilverStripe	514	24	34	26/7	0.22
Smarty	126	13	15	15/0	0.11
SquirrelMail	276	1	1	1/0	N/A
Symfony	2,137	8	8	8/0	0.00
WordPress	387	8	41	0/11	0.22
ZendFramework	4,342	5	5	13/2	0.53

Table 8: Usage of `eval` and `create_function`.

System	Files			VDefs	VCalls	LCalls	Gini
	Total	VA	WI				
CakePHP	640	213	227	36	2,543	830	0.64
CodeIgniter	147	24	26	6	106	106	0.62
DoctrineORM	501	112	112	35	316	303	0.44
Drupal	268	99	108	23	503	268	0.51
Gallery	505	166	170	24	722	199	0.52
Joomla	1,481	999	1,048	15	8,537	419	0.59
Kohana	432	67	67	17	178	88	0.47
MediaWiki	1,480	656	688	90	5,036	1,081	0.63
Moodle	5,367	2,002	2,410	86	11,168	2,716	0.62
osCommerce	529	84	106	0	201	201	0.42
PEAR	74	48	48	1	643	136	0.47
phpBB	269	155	165	6	1,291	973	0.55
phpMyAdmin	341	148	148	5	1,135	858	0.70
SilverStripe	514	328	334	39	994	626	0.54
Smarty	126	26	29	0	109	109	0.53
SquirrelMail	276	74	94	5	388	319	0.64
Symfony	2,137	698	700	38	2,061	1,566	0.49
WordPress	387	287	297	61	5,050	1,513	0.61
ZendFramework	4,342	1,156	1,178	88	4,266	1,014	0.63

Table 9: Usage of Variadic Functions.

files. "N/A" means there was not enough data to compute this—at least two files with occurrences are needed.

From the table, several points are clear. First, overloading is not commonly implemented in typical PHP programs. 3 of the corpus systems do not implement overloading at all, while another 11 define magic methods in fewer than 10 files. The Zend Framework is an obvious outlier, with more magic methods defined in some categories than all the other systems combined, as is Moodle with at least one commonly imported class defining at least one magic method, as can be seen from the *WI* column. Second, the most common magic methods used for overloading appear to be property gets and method calls, with property sets coming in a close third. On the other hand, unset and isset appear to be defined rarely, with static method call support rarer yet—no system in the corpus implements it. Third, definitions of magic methods appear to be distributed fairly evenly, which makes sense intuitively—assumedly, classes that implement `__set` are more likely to implement `__get` (and vice versa).

While this covers definitions of magic methods, it does not cover actual uses of these methods. To detect uses, a type inference analysis of the PHP code is needed.

6.4 The eval Expression

Like many other dynamic languages, PHP provides an `eval` expression that allows arbitrary code, provided as strings, to be executed. As was shown in Table 2, `eval` is not commonly used—in fact, in the entire corpus it appears only 148 times. Similar functionality is provided by the `create_function` function, which, given the function parameters and the function body as strings, creates a uniquely named function and returns it as a value. This appears in the corpus 72 times.

Table 8 shows the distribution of these uses across the various systems in the corpus. Column *Total* shows the total number of files, with *EV* showing the number that have at least one `eval` or `create_function` call. As with overloading, column *WI* then extends this to also consider resolved includes. In most cases this has little to no impact, although in several systems this does show that a significant number of files are impacted through includes. Moodle is especially an outlier here, with 39 files directly containing one of the two features but with the includes analysis revealing there are at least an additional 1038 files impacted. The Total Uses column shows both the number of uses of `eval` (to the left of the `/`) and of `create_function` (to the right). Finally, the Gini coefficient shows how clustered uses of `eval` and `create_function` are in the files where one or both actually appear. For instance, in MediaWiki the 5 uses appear in 5 different files, giving a Gini of 0.00 (perfect equality), while in the Zend Framework 11 of the 15 uses are in the same file, and in CakePHP 4 of the 6 uses are in just a single file.

Unlike with variable variables, discussed above, there are no obvious usage patterns that can be leveraged for static analysis. While two `eval` expressions in CodeIgniter declare a new class using a string literal with no embedded variables, the other `eval` expressions, and all the `create_function` calls, use a combination of embedded variables, string concatenation, and other expressions to build the code string to be evaluated. In some cases it may be possible to ignore `eval` based on how it is used—for instance, MediaWiki appears to only use it in testing and maintenance code, not on the user-facing part of the site. However, given the number of files with at least one of these features in systems like Moodle, it is not possible to assume, in general, that normal user code will not make use of these features. Another option may be to apply static, or hybrid static/dynamic analysis techniques, to try to transform the code into more easily analyzed static code with fewer evals [9, 13, 17].

6.5 Variadic Functions

In PHP, every function is potentially variadic—additional parameters can always be passed to a function, which can ignore them, and it is not an error to pass fewer parameters to a function than it expects (although this does generate a warning). Library functions include a special `...` parameter in the documentation to indicate that they are variadic, while user functions use library functions to process variadic parameters: `func_num_args` to get the number of arguments passed, `func_get_arg` to retrieve a specific argument, and `func_get_args` to get all arguments as an array.

System	Files			CUF	CUFA	CUM	CUMA	Gini
	Total	Inv	Inc					
CakePHP	640	28	34	9	30	0	0	0.17
CodeIgniter	147	6	9	5	3	0	0	0.17
DoctrineORM	501	10	10	3	9	0	0	0.15
Drupal	268	24	40	10	30	0	0	0.30
Gallery	505	20	22	28	23	0	0	0.46
Joomla	1,481	25	30	26	25	0	0	0.41
Kohana	432	8	8	5	7	0	0	0.17
MediaWiki	1,480	89	250	69	80	0	0	0.29
Moodle	5,367	87	1,073	95	69	0	1	0.31
osCommerce	529	2	2	2	0	1	0	0.17
PEAR	74	10	10	20	10	0	0	0.45
phpBB	269	11	44	9	13	0	0	0.31
phpMyAdmin	341	5	5	0	6	0	0	0.13
SilverStripe	514	27	27	32	21	0	0	0.31
Smarty	126	7	8	4	8	0	0	0.29
SquirrelMail	276	3	36	2	2	0	0	0.17
Symfony	2,137	60	62	39	34	0	0	0.15
WordPress	387	39	82	44	50	0	0	0.41
ZendFramework	4,342	92	92	73	86	0	0	0.34

Table 10: Usage of Invocation Functions.

Table 9 shows uses of functions intended to be variadic in the corpus—we do not show calls to functions with too few arguments, or calls where the extra arguments are always ignored. *VA* shows the number of files where a call to a variadic function is made directly, while *WI* shows this number taking account of file inclusion. *VDefs* shows the number of variadic functions declared in the system, determined by looking for calls to the functions used to access the variadic arguments. *VCalls* shows the actual number of calls to variadic functions, while *LCalls* restricts this just to calls of functions from the PHP library, giving an idea of the balance between calls to user-defined variadic functions and library functions. Finally, *Gini* shows the Gini for each system. The Gini is high for each system, meaning that calls are generally grouped in the same files. Given the frequency of these calls, any attempts at inferring the types in a PHP script will need to take variadic functions into account.

6.6 Dynamic Invocation

Along with variable functions and methods, discussed above, PHP provides several functions that allow other functions and methods to be invoked by name: `call_user_func`, `call_user_func_array`, `call_user_method`, and `call_user_method_array` (the latter two deprecated since PHP 4.1.0). Two examples are shown in Figure 6. The first, from MediaWiki, shows the invocation of a static method. On line 3, an array of two elements is created, with the first element holding the name of the class representing the current page, and the second holding the name of the method to invoke on the class, `newFromID`. Line 5 then invokes this method using `call_user_func`, passing the array indicating what to call as the first parameter and the argument to the static method as the second. In the second example (lines 8 through 10) from WordPress, `call_user_func_array` is instead used, with the name of the function, `wp_widget_control`, given as the first argument, and an array containing the parameters to pass to the function given as the second.

Table 10 provides information on uses of these functions in the corpus. Columns *inv* and *inc* are familiar from the prior tables, showing the number of files using at least one of these functions, and then extending this to also account for included files. *CUF*, *CUFA*, *CUM*, and *CUMA* then show usage counts for `call_user_func`, `call_user_func_array`, `call_user_method`, and `call_user_method_array`, respectively. While the first two are used infrequently, the last two, deprecated for years, are almost never used. The last column, *Gini*, again shows the distribution. Unlike with variadic functions, in many cases here the calls are distributed more regularly through the files in which they appear.

```
1  if ($this->mPage->getID() != $this->mRevision->getPage())
2  {
3    $function = array(get_class($this->mPage),'newFromID');
4    $this->mPage =
5      call_user_func($function, $this->mRevision->getPage());
6  }
7
8  $args = wp_list_widget_controls_dynamic_sidebar(
9           array(0 => $args, 1 => $widget['params'][0]));
10 call_user_func_array('wp_widget_control', $args);
```

Figure 6: Dynamic Invocation.

7. CONCLUSIONS

PHP provides a number of dynamic, flexible language features, such as dynamic file inclusion, handlers for unimplemented methods or fields, an `eval` expression for executing arbitrary PHP code at runtime, and variadic functions. Although powerful, features such as these can make programs harder to understand. At the same time, they also make static analysis challenging, hindering development of the code analysis and transformation tools that could most help PHP developers.

As part of our work on developing program analysis tools for PHP, we set out to answer a number of questions about how PHP is used in practice, looking specifically to provide guidance, for ourselves and others, towards building such tools. Using a corpus of large open-source PHP systems, we explored usage of the language in general, showing the distribution of file sizes in the corpus, the distribution of language features in these files, and the frequency with which various language features appear. We also provided information on which features would need to be included in a core language to ensure that most actual PHP files could still be processed, showing that a number of features are used very rarely (in the corpus, sometimes not at all) in real PHP programs.

We then looked specifically at the dynamic parts of the language, investigating how a number of the dynamic features of PHP are used in actual PHP code. We examined dynamic includes, variable constructs, overloading with magic methods, `eval`, variadic functions, and dynamic function and method invocation in our analysis, showing how often these features are used and how they are distributed through the code. We also looked in detail at both dynamic includes and variable variables, showing that many of the former are actually static, while many of the latter are used in patterns that can be exploited by static analysis tools.

We believe there are still additional opportunities for applying empirical analysis such as this to PHP, creating a virtuous cycle where improved analysis leads to more precise empirical results, which can again be used to improve the analysis further. Looking for further usage patterns which can be exploited by analysis tools should be especially fruitful.

8. REFERENCES

[1] Count Lines of Code Tool. `http://cloc.sourceforge.net`.

[2] PHP Language Homepage. `http://www.php.net`.

[3] PHP Usage on GitHub. `https://github.com/languages/PHP`.

[4] PHP Usage Statistics. `http://w3techs.com/technologies/details/pl-php/all/all`.

[5] TIOBE Programming Community Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[6] 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*, pages 397–412. ACM, 2006.

[7] C. S. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

[8] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[9] M. Furr, J. hoon (David) An, and J. S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Proceedings of OOPSLA'09*, pages 283–300. ACM, 2009.

[10] M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static Type Inference for Ruby. In *Proceedings of SAC'09*, pages 1859–1866. ACM, 2009.

[11] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

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

[13] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proceedings of ISSTA'12*, pages 34–44. ACM, 2012.

[14] 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'09*, pages 168–177. IEEE, 2009.

[15] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[16] 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*, pages 105–114. ACM, 2010.

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

[18] 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*, volume 7313 of *LNCS*, pages 104–131. Springer, 2012.

[19] 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'11*, volume 6813 of *LNCS*, pages 52–78. Springer, 2011.

[20] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of PLDI'10*, pages 1–12. ACM, 2010.

[21] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *Proceedings of ECOOP'12*, LNCS, pages 435–458. Springer, 2012.