Exploring PHP Feature Usage for Static Analysis

Mark Hills*, Paul Klint*†, and Jurgen J. Vinju*†
*Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
†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. As with other scripting languages, PHP contains a number of dynamic features that pose challenges to the efficiency and precision of static analysis tools. Because of this, many static analysis tools and techniques applied to other languages have not been applied to PHP.**

In this paper we study how PHP is used in practice. Our goal is to provide the overview and insights needed to steer the design and implementation of static analysis tools for PHP that can be used to detect security vulnerabilities, find other subtle programming errors, and support PHP code refactoring.

We analyze, in general, which features need to be supported to reach reasonable coverage over a representative corpus. More importantly, we focus on language features that pose a challenge to static analysis, and we explore how and when they occur in real programs. Based on this analysis, we recommend several lightweight techniques to mitigate the effect of these features on analysis tools.

I. Introduction

PHP [1], invented by Rasmus Lerdorf in 1994, is a general-purpose programming language focused on server-side application development. PHP is one of the most popular languages for website development, installed on almost 25 million sites [2] and ranking 6th on the TIOBE programming community index [3] as of August 2012. Starting as an imperative language, PHP now includes a single-inheritance class model and features such as interfaces, namespaces, exceptions, traits, and closures.

Like many other scripting languages, PHP is dynamically typed. 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, the strings `"3"` and `"4"`, when added together, yield the number 7; the numbers 3 and 4, when concatenated, yield the string `"34"`; and a call to method `m` is supported by any object that implements `m`, assuming the correct number of parameters is provided.

PHP's flexibility and dynamic nature may sometimes yield unexpected results and make programs hard 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 features provide reflective access to variables, classes, functions, methods, and properties through strings; magic methods allow accesses to either non-existent or non-public methods and properties to be handled on the fly; and the behavior of built-in operations can be puzzling, returning unexpected results (e.g., `"hello"+"world"` is equal to 0). The dynamic nature of PHP programs, in conjunction with the lack of types, provides strong motivation for the construction of *program analysis tools* to aid in program understanding, testing, security vulnerability detection, refactoring, and debugging. Unfortunately, the same features that make it challenging for a human to understand PHP programs also impact the correctness, precision and efficiency of static program analysis tools.

We are interested in answering the following questions:

- Q1 What part of PHP would an analysis tool need to support to cover a significant percentage of real PHP code? See Section V.
- Q2 Where, how often and how are some of the harder to analyze language features used in existing PHP code? See Section VI.
- Q3 Which lightweight techniques can we identify to mitigate the problems caused by these hard to analyze features? See Section VI.

To answer these questions, we have assembled a large corpus of open-source PHP systems, described further in Section III. We then analyzed this corpus using the Rascal [4] meta-programming language (Section IV). Sections V and VI contain our main results and Section VII presents final thoughts and concludes. As is shown in the next section, we are not the first to have done such an empirical study of programs and of how language features are used.

II. Related Work

1) Observing usage: To optimize programming language design, Knuth [5] proposed instrumenting the FORTRAN compiler to acquire usage statistics for language features, and to instrument user code to measure specific statement usage to generate run-time profiles. While CPU profiling is now a widely accepted method, language feature profiling is not. As Knuth aptly states, observing how a language is used could provide valuable insight into what a compiler should actually provide in terms of syntactic constructs, as well as which idioms could be beneficial to focus on for optimization purposes. We share a similar goal in this paper: we want to know how PHP is used to be able to steer the quality of our code processor, which, in our case, is a static analyzer.

Similar work was performed by Morandat et al. [6], who focused on evaluating the design of the R language over a corpus of 3.9 million lines of R code. They employed a

combination of static and trace analysis to gather information on the usage of R language features.

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. [8] also looked at Java bytecode, but focused instead on characterizing the distributions for a number of metrics.

Ernst et al. [9] 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 [10]. Furthermore, Liebig et al. did a targeted empirical study to uncover the use of the preprocessor to encode variations and variability [11]. The current research is executed very much in the same vein, but for PHP, which has its unique properties and problems, different from the C preprocessor and FORTRAN.

2) Analyzing JavaScript: Like PHP, JavaScript is similarly dynamic and reflective in nature. Recent developments in static analysis show that the general solution to dealing with this does not scale very well [12]. In general, a conservative static analysis algorithm for a dynamic language must incorporate all reflective and dynamic features, and include points-to and flow analysis into one great unified fixed point computation with computational complexity in, at least, $\mathcal{O}(n^4)$[1].

On the bright side, Sridharan et al. [12] showed how idiomatic use of these features can be used to greatly improve efficiency and precision of static analyses. They identified the co-existence of reading and writing instructions at the same position in arrays, in which perhaps reflective references are stored. This information about typical usage of JavaScript features enhances the precision of the analysis and greatly reduces the size of the problem to solve.

3) Analyzing PHP: Most work on analyzing PHP programs has focused either on validating the generated HTML or on detecting security vulnerabilities. The latter is often done by performing a *taint analysis*, a form of analysis that ensures that data coming from a client browser, generally either as part of the query string or as form parameters, has been properly checked before being used to construct code run on the server.

Huang et al's WebSSARI [13] uses a combination of static analysis and program instrumentation to protect against security vulnerabilities. The static analysis is an information flow analysis formulated as a type system based on Denning's work on information flow [14] and Strom and Yemeni's work on typestate [15]. An extension of WebSSARI [16] uses bounded model checking techniques to improve error reporting and to provide better instrumentation locations. Jovanovic et al.'s Pixy system [17], [18] uses an interprocedural, flow- and context-sensitive analysis to look for cross-site scripting (XSS) attacks, which inject client-side code into the returned HTML document. Rimsa et al. [19] take a similar approach, focusing on improving the performance of the algorithm used in the analysis. Xie and Aiken [20] use a novel three-tier approach in their analysis to detect SQL injection attacks, using a more detailed analysis at the basic block level which creates summaries that then provide abstractions of program behavior in the intra- and inter-procedural levels.

In Minamide's work [21], the output of a PHP script is represented as a context-free grammar which over-approximates the resulting HTML document. This grammar is then used to check for well-formedness of the generated HTML and to look for cross-site scripting attacks, which can be detected by seeing if strings with problematic constructs are part of the language of the grammar. A similar approach is taken by Wassermann and Su [22], who use context-free grammars to represent sets of string values, finite-state transducers to represent string operations, and regular expressions to represent the security policy being enforced. Samirni et al. [23] use a combined static/dynamic approach to check the HTML generated by a PHP program, ensuring it is not malformed. Instead of using context-free grammars, they use a combination of testing and string constraints.

The PHP-sat [24] and PHP-tools [25] projects include support for security analysis but also add support for additional analyses. These analyses include detection of a variety of common bug patterns (e.g., assigning the result of a function without a return statement) and some PHP4 to PHP5 migration errors, such as name clashes between user-defined and PHP5 library functions. Another tool, the prototype PHP Validator [26], uses a type inferencer as part of a number of possible analyses.

This related work not only shows that static analysis of PHP is relevant but also that a better understanding how PHP features are used in practice may further help the development of such tools.

III. Corpus

The corpus of PHP programs we studied is of course pivotal. Are we interested in *any* kind of PHP code, including one-off experiments and small projects? Or, do we focus on the larger and more widely used software? We chose the latter because (a) such software is most likely to benefit from the use of analysis tools, and (b) such software is larger and therefore not trivial to analyze. We assembled a corpus consisting of releases of the most popular open-source PHP systems, using the rankings given by Ohloh[2], a site that tracks open-source projects, as a proxy for determining popularity. We started by selecting 12 top-ranking systems, extending this to 19 to provide additional diversity of application areas.

Table I shows the most recent release of each system in the corpus[3], which is the version of each system used throughout this paper. *Product* is the name of the system, while *Version* lists the version used. *PHP* and *Release Date* are the PHP version required for the system and the date the given version was released, respectively. *File Count* gives the total number

[1]Here n is the number of syntax tree nodes of the code to be analyzed.

[2]http://www.ohloh.net/tags/php

[3]The assembled corpus consists of multiple versions of each of these systems, going back over a number of years. Newer releases, not yet incorporated into the corpus, may be available for each of these systems.

Product	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 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 I
THE PHP CORPUS: MOST RECENT VERSIONS.

of PHP files (those with `php` or `inc` extensions), while *SLOC* gives the number of PHP source lines totaled across these files. In total, the corpus used in this paper consists of $19,816$ PHP source files with $3,370,219$ lines of PHP source (counted using the cloc [27] tool). *Description* briefly characterizes the product in question.

IV. RESEARCH METHOD

Rascal [4], 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. Using Rascal, we have automated the path from the PHP source code to the tables and figures in the current paper, generating the LaTeX tables and `pgfplots` figures directly using the Rascal string template facility. This ensures that all our experiments, and the resulting tables and figures, are reproducible and checkable. All code is available online at https://github.com/cwi-swat/php-analysis.

We are parsing PHP scripts using our fork [4] of an open-source PHP parser [5], 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.

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

For each file we measure the lines of code and the features it contains. PHP language features – expressions, statements, and declarations – are each represented by one or more AST nodes. For instance, a method definition is represented by a `method` node, while a conditional is represented by an `if` node with optional `elseIf` and `else` nodes. This is the basic data we use for our analysis. It is stored as a map from locations (of PHP files) to a vector of metric values.

Distribution graphs are directly generated from the metric 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 V-B). Computation of feature coverage is a combinatorial optimization problem, which we approximate as described in Section V-C.

The dynamic features we focus on in Section VI are special variants of the features described in Section V, and can be found by looking for specific patterns in the AST. For instance, includes that compute the name of the file to include can be detected by looking for include expression AST nodes with a child node that is not a string literal. The analysis performed to mitigate the impact of these features makes use of Rascal code, while identification of patterns in the code has been performed using a combination of code and manual inspection.

V. PHP IN GENERAL

In this section we provide an overview of the usage of all PHP language features in the assembled PHP corpus. We answer *Q1* —"Which features would a static analysis need to

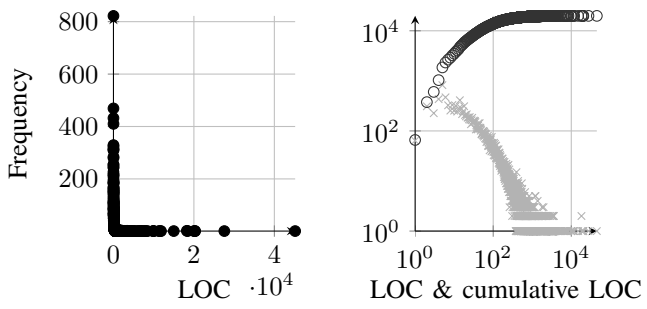

Fig. 1. PHP file sizes histogram on linear and log scale.

cover?"— by exploring the following sub-questions about the corpus:

- How large are PHP files?
- What features do we expect to find in a PHP file?
- What features cover which part of the corpus?

A. How large are PHP files?

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.

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).

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.

B. What features do we expect to find in a PHP file?

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

allocations	array,clone,new,nextArrayElem,scalar
assignments	BitAnd,BitOr,BitXor,Concat,Div,LShift,Minus,Mod,Mul, Plus,RShift,assign,listAssign,refAssign,unset
binary ops	BitAnd,BitOr,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,toObject,toString,toUnset
control flow	break,continue,*declare*,do,exit,for,foreach,***goto***,***haltCompiler***, if,***label***,return,suppress,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,print
unary ops	BitNot,BoolNot,PostDec,PostInc,PreDec,PreInc, UnaryMinus,UnaryPlus

Features in **bold** are not used in the corpus. Features in *italics* and underlined are not needed to achieve 90% and 80% coverage of the corpus,respectively.

TABLE II
LOGICAL GROUPS OF PHP FEATURES USED TO AGGREGRATE USAGE DATA.

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 are typically the "object-oriented" PHP files, with features such as exceptions, interfaces and class definitions, while the other files are mostly programmed in a procedural fashion. 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 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.

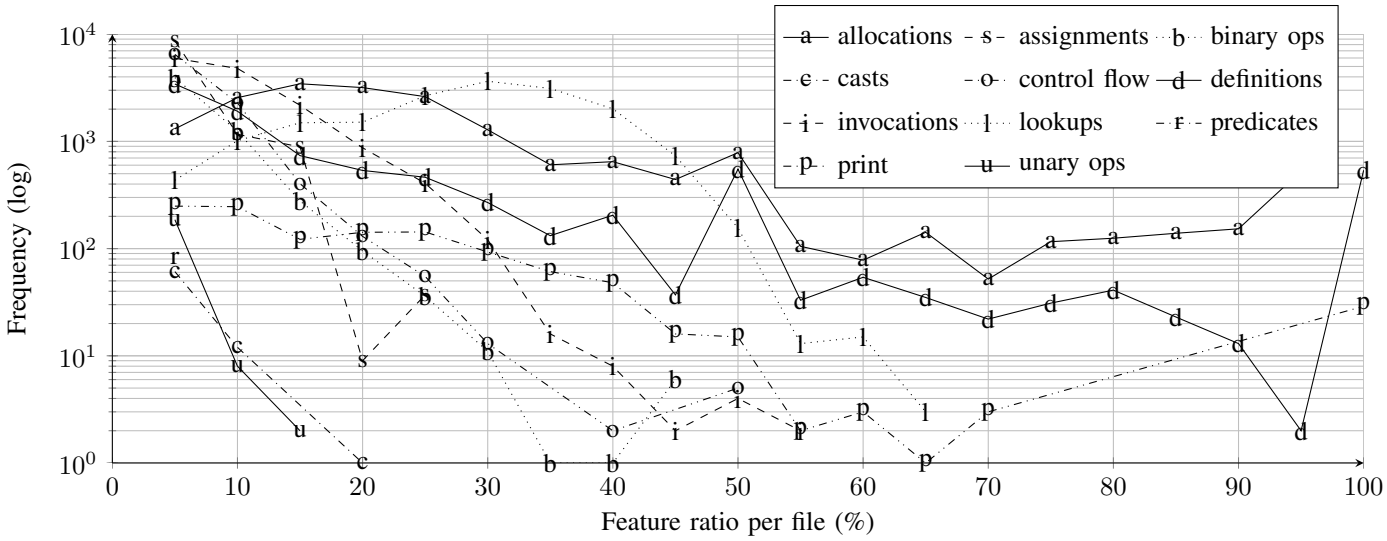

Fig. 2. What features to expect in a given PHP file? This histogram shows, for each feature group, how many times it covers a certain percentage of the total number of features per file. Lines between dots are guidelines for the eye only.

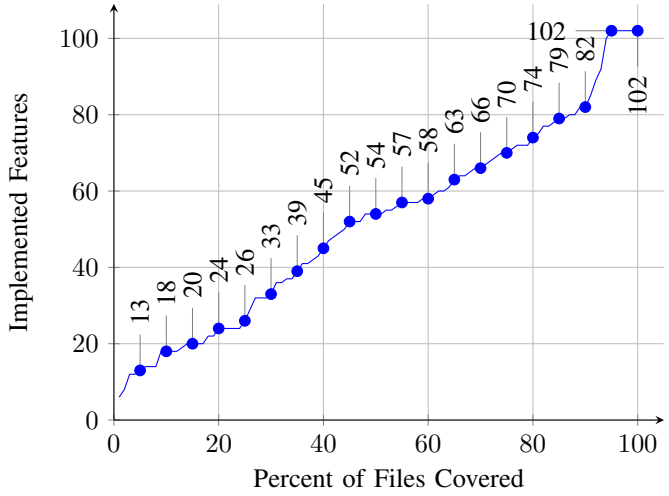

Fig. 3. Features Needed for Percent File Coverage. Numbers of features are shown for each 5% increment in coverage. There are 109 features total.

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: procedural, object-oriented, templates (returning HTML to the browser), and data, where the bulk of PHP files fall into the procedural category.

C. What features cover which part of the corpus?

To build an effective prototype of a static analysis tool, it is important to know which features we need to include. While the information in Section V-B 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?

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 that are sufficient to cover all features used in that percentage of files, with each 5% increment labeled with this number of features. For example, to cover all features used in 50% of the files the analysis tool must implement 54 PHP language features, while 74 features are enough to reach 80% coverage.

Every 10% of added coverage needs about 10 more features

Product	80% set	90% set	Product	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 III
PERCENT OF SYSTEM COVERED BY FEATURE SETS.

implemented. 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 shows which features are sufficient and not which are necessary, it would be surprising if the current graph does not approximate the optimal solution rather closely. More optimal solutions would entail the existence of small groups of features that 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 cover 90% of the corpus 82 of the 102 features are sufficient. The features not needed are shown *in italics* in Table II. The features not needed to achieve 80% coverage are shown underlined in Table II. Using these feature sets, Table III then shows how much of each of the corpus systems would be covered. For instance, the 80% feature set would cover 95.3% of the files in CakePHP.

VI. 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 focus on three of these features: non-literal file includes; variable constructs, which allow string variables to be used in place of identifiers; and overloading, which is used to dynamically handle accesses of undefined or non-visible methods and properties. For each feature, we examine where and how often these dynamic features are used (Q2 in Section I). For the first two, we also examine whether we can identify, and even take advantage of, usage idioms and analysis techniques that would allow us to mitigate the impact of these features on our ability to analyze the code statically (Q3 in Section I). There are a number of other language features which are also challenging to model but which we do not discuss here. This includes `eval`, for evaluating arbitrary PHP code provided as a string, used a total of 148 times in the corpus.

A. Non-Literal Includes

In PHP, a script includes another file using an include expression[6]. The name of the file to include can be provided as a string literal, but can also be given using an arbitrary expression, computed at runtime, and possibly based on user input. In the general case, the file included by an include expression has to be resolved dynamically, and so is not available to static analysis tools. An example, from `includes/Skin.php` in MediaWiki 1.19.1, is 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, based on a similar combination (but with a different string literal) is then included on line 5.

```
1 $deps = "{$wgStyleDirectory}/{$skinName}.deps.php";
2 if (file_exists($deps)) {
3         include_once($deps);
4 }
5 require_once("{$wgStyleDirectory}/{$skinName}.php");
```

Fig. 4. Non-Literal Includes

1) Findings: Table IV provides a high-level overview of the incidence of includes in the corpus. Column *Total* gives the total number of includes, with either literal or non-literal paths, found in each system across all files, with *Non-Literal* then restricting this number to just those includes with non-literal (i.e., computed) paths.

While the includes in the Non-Literal column are all computed, some are easily solvable. For instance, MediaWiki includes a number of include expressions like `require_once("$IP/includes/ProxyTools.php");`. Even though `$IP` is a global variable, there is only one file in MediaWiki that matches the literal part of the path.

To determine which include expressions can be resolved statically, we apply two simplification steps. First, we perform constant propagation and algebraic simplification, specifically for the string concatenation operation, and also perform simplification using some library functions (such as `dirname`) and built-in constants (such as `__FILE__`, a string constant holding the path of the current file) where possible. The results of performing just this step are shown in column *After Simp*. Second, using the expression as a template, we build a regular expression representing the imported path, which is then used to find matching files from the set of all files in the system (assuming that the include points to a valid file, which need not be the case). The results of performing just this step are shown in column *After Match*. Column *After Both* then shows the result of applying the simplification steps first and then performing path matching.

The final three columns provide information about the resulting systems with resolved includes. Column *Resolved%* indicates the percentage of non-literal includes that were resolved over both steps, while column *Files w/Unresolved Includes* shows the number of files remaining that still use non-literal include paths. This doesn't answer whether the remaining non-literal includes are spread evenly throughout these files. Instead, this is shown in the last column, *Gini*, by showing the Gini coefficient calculated over the distribution of non-literal includes in the files that contain them. The Gini

[6]There are four variants: `include`, `include_once`, `require`, and `require_once`, but the distinctions are not important here.

Product	Files	Includes					Resolved%	Files w/ Unresolved Includes	Gini
		Total	Non-Literal	After Simp	After Match	After Both			
CakePHP	640	124	120	29	71	28	76.7%	18	0.29
CodeIgniter	147	69	69	69	53	65	5.8%	22	0.41
DoctrineORM	501	56	54	19	26	19	64.8%	14	0.21
Drupal	268	172	171	169	170	168	1.8%	34	0.56
Gallery	505	44	38	19	37	18	52.6%	12	0.28
Joomla	1,481	354	352	270	327	252	28.4%	206	0.16
Kohana	432	52	48	47	47	47	2.1%	19	0.54
MediaWiki	1,480	554	449	37	332	35	92.2%	25	0.26
Moodle	5,367	7,735	3,917	3,239	3,243	2,716	30.7%	1,151	0.48
osCommerce	529	683	539	151	508	151	72.0%	102	0.22
PEAR	74	211	10	10	10	10	0.0%	8	0.15
phpBB	269	404	385	385	371	371	3.6%	103	0.49
phpMyAdmin	341	811	51	36	48	36	29.4%	26	0.23
SilverStripe	514	373	55	32	36	31	43.6%	11	0.31
Smarty	126	38	36	11	11	11	69.4%	7	0.29
SquirrelMail	276	426	422	117	418	113	73.2%	48	0.42
Symfony	2,137	96	95	60	87	58	38.9%	42	0.23
WordPress	387	589	360	31	241	30	91.7%	18	0.32
ZendFramework	4,342	12,815	350	69	278	68	80.6%	44	0.28

TABLE IV
PHP NON-LITERAL INCLUDES.

coefficient shows inequality, with 0 being complete equality and 1 being complete inequality. Here, larger numbers (such as Drupal, at 0.56) have the remaining non-literal includes spread less evenly throughout the containing files than systems with smaller numbers (such as DoctrineORM, at 0.21).

2) Challenges and Remediation: Non-literal includes provide a challenge for PHP tool writers. If a PHP file contains a non-literal include, to remain sound the analysis would have to assume that any possible includable file could be included, making the analysis both more expensive and less precise. As Table IV makes clear, it is possible to lessen the impact of this problem by performing a lightweight analysis up front, but this still leaves a number of includes that are truly dynamic.

One option to assist analysis tools would be to allow developers to specify, with the include, the files that the include could reference, for instance by using a regular expression. This style of lightweight annotation would allow the programmer intent to be captured without being onerous. A second option would be to use the same techniques that we used above, but to allow sets of files to be derived, versus just one file as is the case now. This would at least limit the file combinations that would need to be handled by the analysis, even in cases where a unique file is not derivable. Finally, in some of the systems we examined, the include path is not truly dynamic, but is built using variables which are treated as constants (or have a very small range of possible values) instead of by using actual constants. By performing additional analysis, it should be possible to determine assignments to these variables that reach the inclusion site. If a set of values reaches the site, these values can be used to reduce the set of possible included files, while if only one value is assigned to the variable we could use the same solution techniques we used for constants.

B. Variable Constructs

PHP includes a number of *variable* constructs, which allow a string variable 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 way (for instance, by including code in a loop) to apply the same functionality over multiple variables, properties, etc.

```
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 }
```

Fig. 5. Variable Variables and Properties.

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

Product	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	35	116	0.44
DoctrineORM	501	0	0	7	15	8	8	5	60	11	21	28	28	108	0.63
Drupal	268	1	1	33	372	2	3	20	91	13	25	50	50	492	0.73
Gallery	505	3	7	3	7	6	14	25	94	13	19	46	46	153	0.52
Joomla	1,481	1	2	6	9	10	11	57	239	45	155	101	104	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	740	2,020	0.59
osCommerce	529	21	89	1	2	0	0	4	7	15	19	38	38	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	47	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	115	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	18	51	0.47
Symfony	2,137	0	0	21	37	20	22	13	98	38	57	89	89	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 V
PHP VARIABLE FEATURES.

array, specifically in the calls to `array_search` and `unset`.

1) Findings: Table V 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 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 column then shows the number of files that include at least one variable feature if we take file inclusion into account. For instance, in MediaWiki 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. This is not an exact count, since this depends on our ability to resolve includes statically (which, as discussed above, cannot be done in general). The final column shows the Gini coefficient, here computed across the distribution of variable features in files that contain at least one. As can be seen from the values in this column, in most systems uses of these features are not distributed equally, but instead tend to cluster in certain files.

From the data, it is clear that uses of PHP's variable features are not rare, but they also do not occur in most of the files in the system. 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: 10 of the systems have a Gini of at least 0.5, while 17 have a Gini of at least 0.4.

2) Looking at Variable Variables: To get a better idea of how variable variables are used in PHP programs, we looked at all 502 occurrences shown in Table V. 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 in one of the following ways: as string literals in an array used in a foreach statement; as a string literal in a conditional or in a case in a switch/case statement; or as a string literal (including from the above two 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, 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.

The summary of our findings is shown in Table VI. The product 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 V. The second and third columns show the number of cases where the set of variable names is statically computable, and the cases where it 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 the variable variables are statically determinable. Many of these cases are to remove repetition of code. 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.

Product	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 VI
DERIVABILITY OF VARIABLE-VARIABLE NAME ASSIGNMENTS.

Product	Files		Magic Methods						Gini
	MM	WI	S	G	I	U	C	SC	
CakePHP	18	18	5	12	7	0	10	0	0.28
CodeIgniter	4	4	1	5	0	0	1	0	0.32
DoctrineORM	4	4	1	1	0	0	3	0	0.15
Drupal	2	2	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	98	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 VII
PHP OVERLOADING (MAGIC METHODS).

3) Challenges and Remediation: 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.

There are several options for lessening the impact of PHP's variable features. First, as we showed above, a combination of reaching definitions and algebraic simplification over string operations can sometimes resolve a number of the uses to individual strings (e.g., based on a case branch) or small sets of strings (e.g., an array provided as part of a `foreach` loop). Improving this analysis, for instance to show that a called method does not mutate the value assigned to a variable passed to it as a parameter, would allow the analysis to resolve even more of the names. A matching technique, similar to what we used on include names, could also work in cases where we have the partial name of a variable, property, class, etc.

Another option is to allow for annotations to be placed at strategic locations in the code (e.g., on function or method signatures). In some cases it was possible for us to identify the strings that would be live at the point of use based on comments in the source, but this information is not available to tools, and is not actually enforced by the program. A lightweight annotation mechanism in source comments, based on something familiar like regular expressions, would allow for this information to be used in the analysis, and would provide a way to generate code that would enforce it as well if desired.

Finally, we expect that the availability of closures will reduce the use of variable features, at least in situations where they are used to avoid duplicating code. Closures would most likely not reduce some of the more complicated uses of variable features tied to reflection.

C. 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.These magic methods 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 VII provides an overview of the use of overloading 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. 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 files. "N/A" means there was not enough data to compute this–at least two uses are needed.

From the table, several points are clear. First, overloading is not a commonly used feature in typical PHP programs. 3 of the corpus systems do not use it at all, while another 11 use it in fewer than 10 files. Second, the most common uses 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 used rarely, with static method call support rarer yet – no system in the corpus uses it. Third, uses of these features appear to be distributed fairly evenly, which makes sense – assumedly, classes that implement `__set` are more likely to implement `__get` (and vice versa). We have not looked further to see if this relationship holds between

magic methods used to implement properties and those used to implement method calls.

VII. Conclusions

Our findings can be summarized as follows:

Q1. What part of PHP would an analysis tool need to support to cover a significant percentage of real PHP code? Our frequency data show what features to expect in a given PHP file, which can also help in selecting test cases from the current corpus for a given static analysis tool. For prototyping and evaluation, being able to deal with file sizes up to 1300 LOC is enough to handle a significant number of files. Also, not all features need to be implemented immediately: 74 features cover 80% of the corpus, while 82 cover 90%, and 7 are never used (Table II). Since most PHP code is procedural, it is possible to initially de-emphasize the object-oriented features.

Q2. Where, how often and how are some of the harder to analyze language features used in existing PHP code? They are used sporadically as compared to other language features, and their usage is often concentrated in a small set of files. We found indications that the impact of variable features and overloading is not spread much via file inclusion (using an incomplete view of the include dependencies). We found that lightweight static analysis techniques help in an encouraging number of cases.

Q3. Which lightweight techniques can we identify to mitigate the problems caused by these hard to analyze features?

As shown in Section VI, a significant number of non-literal includes can be resolved with a combination of constant propagation, algebraic simplification, and regular expression matching over file paths. Adding a reaching definitions analysis would allow variable parts of the path names to be resolved in some additional cases. Also, allowing lightweight annotations with path regular expressions, and allowing the current analysis to return sets of possible results, would both help in cases where a unique file name either cannot be statically found or is not intended. With PHP's variable features, options for improvement include matching over the literal parts of computed variable names, lightweight annotations, and improvements in the current analysis (a combination of reaching definitions and algebraic simplification). An encouraging number of variable variables, 61% (77% for PHP5 systems), can already be resolved, and this should improve the number further. Converting existing code to using closures, in cases where this is possible, would remove many of the simple "code-saving" usage scenarios as well.

While answering these questions, we have characterized the usage of programming language features in a large corpus of PHP files. Our fully reproducible experimental analysis reveals insights that are compatible with the opinions of PHP compiler writers with deep PHP knowledge (e.g., [28]). We produced hands-on material for making engineering decisions, facilitating the creation of novel static analysis tools. Finally, we identified several lightweight resolution techniques for PHP's dynamic features to help mitigate their impact on efficiency and accuracy of analysis tooling.

References

[1] "PHP Language Homepage," http://www.php.net.

[2] "PHP Usage Statistics," http://trends.builtwith.com/framework/PHP.

[3] "TIOBE Programming Community Index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[4] 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*. IEEE, 2009, pp. 168–177.

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

[6] 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.

[7] 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.

[8] 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'10*. ACM, 2006, pp. 397–412.

[9] 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.

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

[11] 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.

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

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," in *Proceedings of WWW'04*. ACM, 2004, pp. 40–52.

[14] D. E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[15] R. E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 157–171, 1986.

[16] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo, "Verifying Web Applications Using Bounded Model Checking," in *Proceedings of DSN '04*. IEEE, 2004, pp. 199–208.

[17] N. Jovanovic, C. Krügel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," in *IEEE Symposium on Security and Privacy*, 2006, pp. 258–263.

[18] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities," in *Proceedings of PLAS'06*. ACM, 2006, pp. 27–36.

[19] A. Rimsa, M. d'Amorim, and F. M. Q. Pereira, "Tainted Flow Analysis on e-SSA-Form Programs," in *Proceedings of CC'11*, ser. LNCS, vol. 6601. Springer, 2011, pp. 124–143.

[20] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," in *Proceedings of the 15th USENIX Security Symposium*, 2006, pp. 179–192.

[21] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," in *Proceedings of WWW 2005*. ACM, 2005, pp. 432–441.

[22] G. Wassermann and Z. Su, "Static Detection of Cross-Site Scripting Vulnerabilities," in *Proceedings of ICSE'08*. ACM, 2008, pp. 171–180.

[23] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren, "Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving," in *Proceedings of ICSE'12*. IEEE, 2012, pp. 277–287.

[24] "PHP-Sat Project," http://www.program-transformation.org/PHP/PhpSat.

[25] "PHP-Tools Project," http://www.program-transformation.org/PHP/PhpTools.

[26] P. Camphuijsen, "Soft typing and analyses of PHP programs," Master's thesis, Universiteit Utrecht, 2007.

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

[28] P. Biggar, "Design and Implementation of an Ahead-of-Time Compiler for PHP," Ph.D. dissertation, Trinity College Dublin, April 2010.