View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Static, lightweight includes resolution for PHP

Mark Hills, Paul Klint, Jurgen Vinju

To cite this version:

Mark Hills, Paul Klint, Jurgen Vinju. Static, lightweight includes resolution for PHP. ASE 29 Proceedings of International Conference on Automated Software Engineering 2014, Sep 2014, Vasteras, Sweden. pp.503 - 514, 10.1145/2642937.2643017 . hal-01110903

HAL Id: hal-01110903

https://hal.inria.fr/hal-01110903

Submitted on 29 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Static, Lightweight Includes Resolution for PHP

Mark Hills[1]
[1]East Carolina University
Greenville, NC, USA

Paul Klint[2], and Jurgen Vinju[2,3]
[2]Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[3]INRIA Lille Nord Europe, Lille, France

ABSTRACT

Dynamic languages include a number of features that are challenging to model properly in static analysis tools. In PHP, one of these features is the `include` expression, where an arbitrary expression provides the path of the file to include at runtime. In this paper we present two complementary analyses for statically resolving PHP includes, one that works at the level of individual PHP files and one targeting PHP programs, possibly consisting of multiple scripts. To evaluate the effectiveness of these analyses we have applied the first to a corpus of 20 open-source systems, totaling more than 4.5 million lines of PHP, and the second to a number of programs from a subset of these systems. Our results show that, in many cases, includes can be either resolved to a specific file or a small subset of possible files, enabling better IDE features and more advanced program analysis tools for PHP.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Measurement, Experimentation

Keywords

Static analysis, dynamic language features, PHP

1. INTRODUCTION

PHP,[1] 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 April 2014 ranking 7th on the TIOBE programming community index,[2] used by 81.9 percent of all websites whose server-side language can be determined,[3] and ranking as the 4th most popular language on GitHub by repositories created in 2013.[4] PHP is dynamically typed, with a single-inheritance class model (including interfaces) 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"`. Along with dynamic types, PHP includes a number of dynamic language features also found in other dynamic languages such as Ruby and Python, including an `eval` expression to run dynamically-built code and special methods (referred to as *magic methods*) that handle accesses of object fields and uses of methods that are either not defined or not visible.

Although the popularity of PHP should have resulted in a plethora of powerful development tools, so far this has not been the case. We believe this is because the same dynamic features that make PHP so flexible also make it challenging for static analysis. This hinders the creation of the static analyses needed as the foundation on which these programmer tools are built, and also causes challenges for static analysis tools used for tasks such as security analysis, a topic we touch on briefly in Section 6.

In this paper, we focus on one specific example of a dynamic feature which is problematic for static analysis: the PHP file inclusion mechanism. In PHP, like in many other dynamic languages, file inclusion is performed dynamically—the file to be included is specified using an arbitrary expression which, at runtime, evaluates to a file path. The file at this path is then loaded, with some parts of the file brought in as top-level definitions (e.g., classes and functions) and other parts inserted directly at the point of the call and executed. This provides an obvious challenge for static analysis—it may not be possible to even determine the source code that needs to be analyzed until runtime.

Is it really the case, though, that these dynamic includes are truly dynamic, or is it possible to resolve many of them

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

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[3] `http://w3techs.com/technologies/details/pl-php/all/all`

[4] The query for this is based on `http://adambard.com/blog/top-github-languages-for-2013-so-far/`, but including the entire year.

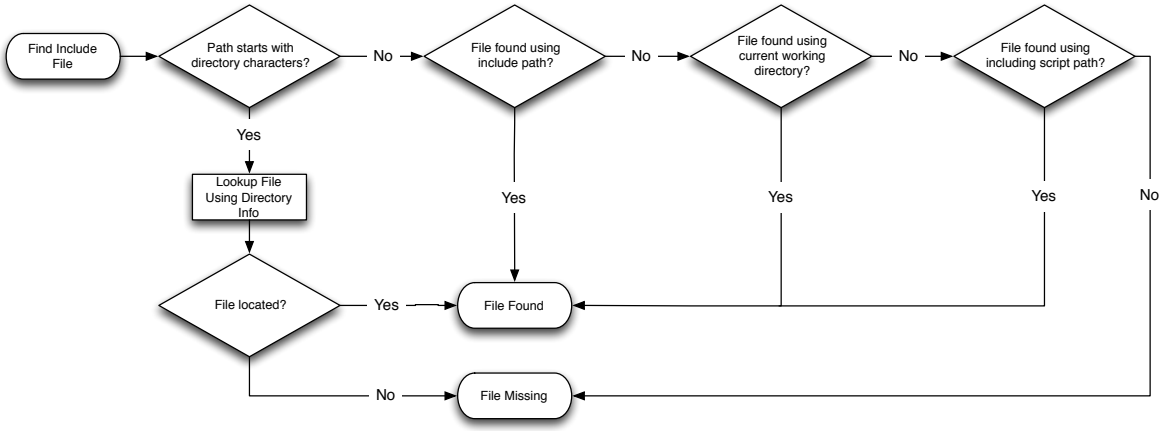

Figure 1: The PHP File Inclusion Process.

statically? In prior work [6], we showed that the latter is actually the case: many apparently dynamic includes are in practice static. String building expressions used in these includes are not often used in a truly dynamic fashion, where the file that is included depends on a dynamic value. Instead they are used for the mundane task of navigating the local file system towards a single file, taking advantage of system wide constants and functions for string concatenation and directory navigation.

The main contributions presented in this paper are as follows. First, taking advantage of the findings of our prior work, we have developed two includes resolution algorithms. The first analysis, dubbed FLRES, looks at a specific PHP file in isolation, quickly determining which files could be included by each `include` expression in that file. This analysis works with any PHP file in the system, and is intended to provide information for more precise analysis as well as support for IDEs. The second analysis, dubbed PGRES, looks at a specific PHP program—a page loaded from a server, or a command-line program—and attempts to resolve all includes for this program, even those that are loaded transitively by already included files. This analysis is more precise, since the context of the initial `include` expression is sometimes required for correct resolution, and also models the possibility that the file cannot be determined statically, such as in cases where includes are used to load plugins.

To the best of our knowledge, these are the first published algorithms for the static resolution of dynamic includes in PHP. Determining the actual file or files that can be included at runtime allows static analysis tools to get access to more of the program source that will be executed, opening up possibilities for more powerful analysis techniques to be applied [18] and enabling more powerful developer tools, including tools for finding errors, performing code refactoring, detecting security violations, and supporting standard IDE services such as "Jump to definition".

Second, to ensure these algorithms works in practice, we have validated them empirically. FLRES has been validated against a corpus of 20 open-source systems, made up of 32,682 source files and 4,593,476 lines of code, while PGRES has been validated against a number of programs from these systems. This corpus includes a number of the most used PHP systems, including well-known systems such as WordPress, Joomla, and MediaWiki as well as frameworks such as Symfony.

The rest of the paper is organized as follows. In Section 2, we discuss PHP includes in more depth, describing the method used by PHP to resolve includes at runtime, showing examples of both static and dynamic includes, and describing which cases we focus on in this paper. Following this, in Section 3 we present FLRES, with PGRES presented in Section 4. To show that the analyses work in practice, Section 5 evaluates the effectiveness of these algorithms using the corpus mentioned above. Finally, Section 6 describes related work, and Section 7 concludes. All software used in this paper, including the corpus used for the validation, is available for download at `https://github.com/cwi-swat/php-analysis`.

2. OVERVIEW

Here we explain how includes work in PHP, the technical context of our work, and the necessity of creating two distinct algorithms for resolving includes.

2.1 PHP Include Semantics

In PHP, `include` expressions are used to include the contents of one file into another at runtime. We list here what the PHP interpreter normally does dynamically, and what our two algorithms approximate statically in a conservative fashion. The `include` expression takes a single operand; this operand is expected to evaluate to a string containing the file name, possibly including some information about the path to the file. This operand can either be given *statically*, as a string literal, or *dynamically*, using an arbitrary expression that yields a string.

Once the operand is evaluated, the include mechanism associates the string result (referred to below just as the *file name*, although it may include path information as well) to an actual file (referred to here as the *target file*), present either in the system or in installed libraries. This file resolution is normally done by the PHP interpreter according to the dynamic process depicted in Figure 1. If the file name starts with "directory characters", e.g., `\`, `/`, `.`, or `..`, the target file is looked up using this directory information. File names beginning with `\` or `/` are absolute paths, looked up from the root (of the site for websites, or the file system for command-line tools), while names beginning with `.` or `..` are relative

```
// 1. /includes/normal/Utf8Test.php, line 30
require_once 'UtfNormalDefines.php';

// 2. /includes/UserMailer.php, line 240
require_once( 'Mail.php' );

// 3. /maintenance/showStats.php, line 31
require_once( dirname(__FILE__)
  . '/Maintenance.php');

// 4. /tests/jasmine/spec_makers/makeJqueryMsgSpec.php,
// lines 17 and 19
$maintenanceDir = dirname(dirname(dirname(dirname(
    dirname(__FILE__))))).'/maintenance';
require( "$maintenanceDir/Maintenance.php" );

// 5. From /includes/Skin.php, lines 151 to 154
$deps = "{$wgStyleDirectory}/{$skinName}.deps.php";
if (file_exists($deps)) {
        include_once($deps);
}
```

Figure 2: Includes in MediaWiki 1.19.1.

paths, looked up starting in the directory containing the script currently being executed.

Otherwise, the target file is computed using the following process. First, the file name is looked up starting from each directory given as part of the include path. The lookup stops when the target file is found, even if other, later entries in the include path have not been tried. The include path can hold either absolute or relative paths—although it is discouraged, the current directory `.` is often given in the include path. Next, the current working directory, which may be different from the directory containing the executing script, is checked for the file. Finally, an attempt is made to find the target file by looking it up in the directory containing the executing script. If the file is not found in any of these locations, or if the lookup based on absolute or relative paths discussed above fails to find the file, the process ends in the "file missing" node, meaning the include cannot be performed.

In summary, the semantics of dynamic includes are affected by the given file name, the include path, and the location and working directory of the script, and the file system is searched in a well-defined order for the file being included.

When the `include` expression is evaluated, any functions, classes, interfaces, and traits in the target file are included into the current global scope. The other contents of the included file are executed as part of evaluating the expression, and inherit the current execution environment, meaning they run as if they were inserted at the spot of the original `include`. A `return` statement evaluated in the included code will immediately return back to the point of the original `include` expression, skipping the rest of the code that would normally be evaluated. The value of an `include` expression is `FALSE` in case of failure, and either 1 or the explicitly returned value in the case of success.

If the file to be included does not exist, a standard `include` expression will issue a warning, while a variant, `require`, will produce a fatal error. If the file to be included has already been included, it will be included again unless the `once` variants—`include_once` and `require_once`—are used. These variants ensure that the contents of a file are included at most once. In the remainder of this paper we do not bother to distinguish between these variants since they do not influence the algorithm.

Several example `include` expressions from MediaWiki 1.19.1 are shown in Figure 2. In the first example, the included file, `UtfNormalDefines.php`, is given statically, and is found in the same directory as the file with the `include` expression, `Utf8Test.php`. Since no path information is given, this file will be found by looking in the include path (if one is given) and then in the directory of the including script, by which point it will definitely be found. In the second example, the included file, `Mail.php`, is also given statically. However, `Mail.php` is not part of MediaWiki, but is instead part of the PEAR Mail package, and is found by looking on the include path.

We classify the remaining three examples, and others like them, as dynamic includes. In the third example, `__FILE__` is a so-called "magic constant"; `__FILE__` evaluates to the full path, including file name, of the file which uses it. `dirname` is a standard library function which returns the parent directory of a file. Taken together, this will evaluate to the absolute path of the `/maintenance` directory, with the string concatenation operation (i.e., the dot operation) then adding the file `Maintenance.php` to the end of the path. The fourth example provides an alternate way of looking up the same file: the starting directory is more deeply nested, so more calls to `dirname` are used, with the result assigned to variable `$maintenanceDir`. This is then used inside a string built using string interpolation (the variable is replaced by its value) and the name of the `Maintenance.php` file. Finally, in the fifth example, the location of the file is built as a string expression with two variables, one for the MediaWiki style directory and one for the name of the "skin" used to customize site appearance. The resulting file path is saved into a variable; this variable is checked to see if the file exists and, if so, the file is included. Of these three dynamic include examples, we are currently able to resolve the first two to unique target files using the techniques described below. The last is truly dynamic, and cannot be resolved to a single file—the best that can be done is to partially resolve it to those files ending in `deps.php`.

2.2 The PHP AiR framework

To statically resolve such dynamic includes we analyze PHP using Rascal as part of our ongoing work on PHP AiR, a framework for PHP Analysis in Rascal [5]. Rascal [10] is a meta-programming language for source code analysis and transformation that 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, and its string templating and IO facilities, to generate the LaTeX for the tables in this paper that document the corpus and the analysis results. All code is available online at `https://github.com/cwi-swat/php-analysis`.

We are parsing PHP scripts using our fork[5] of an open-source PHP parser[6], 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.

2.3 The Need for Two Algorithms

In this paper we present two includes resolution algorithms, one for individual files and one for programs. The first,

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

Input : *sys*, a PHP system, a mapping from file locations to abstract syntax trees
Input : *toResolve*, a location indicating the AST in *sys* to be analyzed
Input : *baseLoc*, a location indicating the root of system *sys*
Input : *libs*, a set of locations of known library includes used by *sys*
Output : A relation *res*, from the location of each include expression to the location(s) of possibly included files

```
1  iinfo ← include info cache for sys
2  ast ← the AST for file toResolve in sys
3  includes ← a set containing each include expression in ast
4  for each i ∈ includes do
5      i ← normalizeExpr (replaceConstants (i,iinfo))
6      iloc ← the location of i
7      ip ← the operand used in i indicating the file to include
8      if ip is a literal string starting with / then
9          if sys contains a file at baseloc + ip then
10             add iloc × (baseloc + ip) to res
11             remove i from include, continue with next i
12         end
13     end
14     for each file f returned by matchIncludes (sys,i,baseLoc,libs) do
15         add iloc × (baseloc + ip) to res
16     end
17 end
```

Algorithm 1: PHP File-Level Includes Resolution Analysis (FLRES)

FLRES, for individual files, computes files that could be included directly by a given file, and does not consider the *context* of the `include`—the directory of the original script being executed, the current working directory, or the include path. The second, PGRES, for programs, is "context-sensitive", considering all this information in order to more precisely resolve the includes in the original file as well as those brought in as part of the includes process.

Figure 3 illustrates one typical scenario where this distinction is critical. Here, two files, `/A.php` (for brevity, just A below) and `/admin/B.php` (B), both include file `/includes/C.php` (C). C includes file `D.php` (D), but does not provide any path information. When looking at C in isolation—*without knowing it is being included by another file*—the analysis should indicate that either the file `/D.php` (D_1) or the file `/admin/D.php` (D_2) could be loaded by the `include`. Note that here this is not an over-approximation, as each file could be loaded by this `include` expression.

The program-level analysis, PGRES, will instead look individually at both A and B, two programs—scripts that can be invoked directly—and will determine which files could be included when either is executed. When C is included in A, the code contained in C will be executed in the context of A. When the `include` expression that includes D is executed, this means it will not run in the context of C—in the `/includes` directory—but will instead run in the context of A. Since D_1 is in the same directory as A it will be included (following the solid line), and the ambiguity over which version of D to include disappears. The same happens with B—when C is included, the code in C runs in the context of B, with D_2 found in the same directory as B (the dashed line), again making the file to be selected unambiguous.

The file-level includes analysis, FLRES, is described next in Section 3. The program-level includes analysis, PGRES, is described later in Section 4.

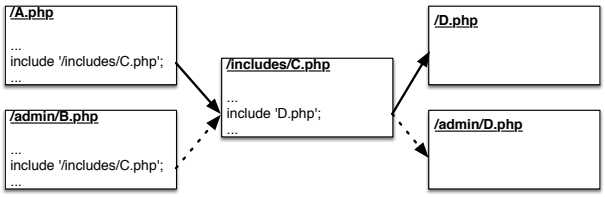

Figure 3: An Ambiguous File-Level Include.

3. FILE-LEVEL INCLUDES RESOLUTION

The FLRES algorithm (see Algorithm 1) takes as input a representation of the PHP system under analysis (*sys*); the location of the PHP file to analyze (*toResolve*); the root location of the system under analysis (*baseLoc*); and in *libs* the locations of all standard libraries used by *sys*. The *sys* representation is a map from file names to ASTs for each file (see Section 2). We need to assume that *libs* is a correct and complete list of library dependencies; *libs* can be empty if the system does not use any external libraries (those that come with PHP, such as the standard MySQL libraries, do not need to be explicitly included to use). *iinfo* is a cache, similar to that maintained by IDEs, containing information about the system under analysis, such as the locations of defined constants. Currently extracted using Rascal, we plan to integrate this with Eclipse to allow it to be updated incrementally as files are edited.

The algorithm collects all instances of include expressions in the AST for a script. It then performs two main steps to find the files that could be included by each. First, the call to `replaceConstants` replaces each constant in the include expression by the value of the constant, under the conditions that the constant is defined using a literal value, or an expression that can statically be converted to a literal value, and that all definitions of the constant in *sys* are identical.

Second, the call to `normalizeExpr` performs a number of

Constant Name	Replacement Value
__FILE__	Absolute path including file name
__DIR__	Absolute path, without file name
__CLASS__	Name of the enclosing class
__METHOD__	Name of the enclosing method
__FUNCTION__	Name of the enclosing function
__NAMESPACE__	Name of the enclosing namespace
__TRAIT__	Name of the enclosing trait

Table 1: Magic Constants in PHP.

simplifications, including simulating the effects of common functions for working with strings and directory names, replacing magic constants with their values (shown in Table 1), and converting concatenations of string literals into a single literal. This simplification process continues until no more changes to the expression being simplified are found. A common example of how magic constants are used in include expressions in combination with builtin functions is shown in Figure 2 in the third example, where `__FILE__` is used in conjunction with the `dirname` function. This algebraic simplification process is run repeatedly to take advantage of new opportunities for replacements, until no more constant expressions can be collapsed into literal expressions.

At this point, if the expression indicating what to include (given as *ip*) is a literal string starting with `/` (or `\`), an attempt is made to find the file in *sys* by adding *ip* to the end of *baseloc*. If this file is found, we consider the include to be resolved and add this mapping to *res*, which is a relation from include locations to the locations of the files that could be included at each location. We also skip the rest of this iteration of the loop. If *ip* is not a literal string starting with `/`, or if the file lookup failed, FLRES instead attempts to use regular-expression matching to find possible matches, converting *ip* into a regular expression and returning all files in *sys* and in *libs* that match. In some cases, where *ip* is an arbitrary expression, this could be all the files in both. A pair, consisting of the location of *i* (given as *iloc*) end each returned file *f*, is then added into *res*.

Soundness: FLRES provides a sound over-approximation of the files that could be included by each include expression in a file under two conditions. The first condition is that the files given in *sys* and *libs* are actually all the files that could be included. If, for some reason, additional libraries are being used but are not included in *libs*, these files will be missed. Files included using `eval` are also not detected (although we are unaware of any code in the corpus that actually does this). The second condition is that the constants used in an include expression are actually visible at that point in the code. For instance, when given an include path like `ROOT . 'myfile.php'`, if there is a unique constant `ROOT` defined as `'/base/'` FLRES assumes this constant is being used. However, if a constant is not visible, the name of the constant is instead treated as a string literal, meaning the above should attempt to include file `ROOTmyfile.php`. We are unaware of any cases where this specific scenario occurs in practice (it is more likely that the include would just fail), but it is a theoretical possibility.

In summary, FLRES is a fast, lightweight algorithm that is based on straightforward algebraic simplification and constant propagation. In Section 5 we will evaluate how accurately it performs on real PHP systems.

4. PROGRAM-LEVEL RESOLUTION

The PGRES algorithm (see Algorithm 2) receives six inputs. The first four are the same inputs given to FLRES. The fifth parameter, *ipath*, represents the include path for PHP, and is given as a list of locations. To remain sound, the algorithm makes few assumptions about the include path; PGRES assumes that the given include path is accurate, but additional entries may be present if language features that can alter the include path are reachable, something checked in the body of the algorithm. The sixth argument, *flres*, is the output of FLRES, which serves as a starting point for PGRES. Part of this information is the include expression, with the `replaceConstants` and `normalizeExpr` steps already applied, meaning that this does not need to be done again here. The *iinfo* on line 1 is the same cached system information used in FLRES.

On line 2, the initial include graph, *igraph* is constructed, using the already-computed information from FLRES in *flres*. Starting from *toResolve*, a node is created for each script in *sys* and each library in *libs* that is reachable based on the (over-approximation) of what is included by each `include` expression as given in *flres*. Edges are then added to mirror this relation. Each edge includes the `include` expression that induced it, again based on the already-simplified expression stored in *flres*, and also tracks the (unique) source node and 0 or more target nodes.[7]

On line 3, *setsIP* is computed. This set contains the locations of all reachable scripts that may set the PHP include path or the PHP working directory at runtime. The include path can be changed by calling the `set_include_path` function or the `ini_set` function (although, for caveats, see the discussion of soundness below) and supplying the new include path. In both cases, the include path is changed for the remainder of the script. In the case of `ini_set`, to maintain soundness, we assume the call *could* set the include path unless a string literal is used to name a different setting. The working directory is modified by calling `chdir`. Once this is done, line 4 then annotates all the nodes of the include graph with information on the related script, specifically which constants it defines and whether it sets the include path or changes the working directory (i.e., is in *setsIP*).

After this, the loop on lines 5 through 18 continues while the edge set of the include graph changes. In the first step, on line 6, the include expressions on each edge are simplified according to what is currently known about the includes relation. This is similar to the simplification steps described for FLRES, but also takes advantage of information about which constant definitions are actually reachable. The same constant can be given different values in different scripts, which does occur in practice, but it may be the case that only one definition of the constant is actually reachable. This allows additional constant replacements beyond what is possible in FLRES for the situations when the constants are not unique throughout the entire system.

The inner loop, on lines 7 through 17, then iterates over each edge *e* which does not already have a unique target file, with the goal of updating the target node sets to account for

[7]This means we do not create one edge for each possible source/target combination, which reduces the number of edges and simplifies the algorithm.

```
Input  : sys, a PHP system, a mapping from file locations to abstract syntax trees
Input  : toResolve, a location indicating the AST in sys to be analyzed
Input  : baseLoc, a location indicating the root of system sys
Input  : libs, a set of locations of known library includes used by sys
Input  : ipath, the include path, a list of locations
Input  : flres, file-level resolve information from FLRES
Output : A relation res, from the location of each include expression to the location(s) of possibly included files
1  iinfo ← include info cache for sys
2  igraph ← build initial include graph based on FLRES results
3  setsIP ← reachable scripts that set the include path or change the working directory
4  igraph ← annotateNodes (iinfo,setsIP)
5  while the set of edges in the include graph changes do
6      igraph ← simplifyEdges (igraph,iinfo)
7      for e a non-unique edge do
8          i ← the include expression for edge e
9          ip ← the operand used in i indicating the file to include
10         if ip is a literal string then
11             changesIP ← a file that sets the include path or changes the working directory is reachable
12             e ← calculateLoc (toResolve,baseLoc,ipath,changesIP)
13         end
14         for each file f returned by matchIncludes (sys,i,baseLoc,libs) do
15             add iloc × (baseloc + ip) to res
16         end
17     end
18 end
19 res ← the relation, from include expressions to files, induced by igraph
```

Algorithm 2: PHP Program-Level Includes Resolution Analysis (PGRES)

any new information. If the file name given as an operand to the `include` expression (assigned to *ip*) is a literal string, `calculateLoc` is called to compute the actual include file represented by the location.

`calculateLoc` identifies the file to be included by walking a tree model of the files in *sys*, following the procedure laid out in Figure 1. This also provides an increase in precision over FLRES, since, unlike FLRES, PGRES knows the include path and the directory of the executing script. Before the call to `calculateLoc`, *changesIP* is first set to true if a script that sets the include path or changes the working directory is reachable in the include graph. If *changesIP* is true, it will "short circuit" the process shown in Figure 1—since it then may be the case that any directory is on the include path, or any directory is the current directory, if no directory characters are found at the start of the file name `calculateLoc` will fail, leaving the information about the target files for this include unchanged.

If, after this, edge *e* can still point to multiple nodes, the same matching process used in FLRES is used here as well. This is repeated since, as the number of reachable constants decreases, more constant replacements may become available; this allows the file names used in the matching process to improve, making the match more precise over time.

On line 19, once no more changes occur in the include graph, the result, *res*, is computed as the relation, from include expressions to included files, induced by *igraph*.

Soundness: PGRES provides a sound over-approximation of the files that could be included by each include expression in a program under several common conditions. The first two are identical to those for FLRES, and are discussed in Section 3. Additionally, PGRES also assumes that the include path and the working directory are not changed in obfuscated ways, e.g., by invoking `ini_set` using a variable function or inside `eval`. If this is done, it could cause the wrong file to be found during the lookup process in `calculateLoc`. We believe this would be highly unusual, though, and are not aware of any real PHP code that actually does this.

5. EVALUATION

In this section we present the results of our evaluation of the two includes resolution algorithms—FLRES and PGRES—described, respectively, in Sections 3 and 4. First, we describe the systems used in the evaluation. Then, the two algorithms are evaluated separately. First FLRES, the file level resolution algorithm, is evaluated on the entire corpus to measure its capability of resolving includes to small sets of possible PHP files. Then we evaluate the improvement of PGRES over FLRES by applying it to a subset of the corpus focusing just on actual PHP programs—scripts that can be executed directly from a web server or the command line.

5.1 Corpus

As part of our prior work on PHP [6], we assembled a corpus of 19 large open-source PHP systems, basing our choice on popularity rankings provided by Ohloh[8], a site that tracks open-source projects. We have since extended this with an additional system—Magento, a popular online retail system—and updated all systems in the corpus to more recent versions if available. The chosen systems are shown in Table 2. Systems were generally selected just based on the Ohloh ranking, although in some cases we skipped systems if we already had several, more popular systems of the same

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

System	Version	PHP	Release Date	File Count	SLOC	Description
CakePHP	2.4.4	5.2.8	12/24/13	661	148,335	Application Framework
CodeIgniter	2.1.4	5.1.6	7/8/13	147	24,382	Application Framework
Doctrine ORM	2.3.3	5.3.3	5/11/13	609	49,126	Object-Relational Mapping
Drupal	7.24	5.2.5	11/20/13	274	89,266	CMS
Gallery	3.0.9	5.2.3	6/28/13	505	39,087	Photo Management
Joomla	3.2.1	5.3.1	12/18/13	2,117	221,208	CMS
Kohana	3.3.1	5.3.3	9/1/13	468	29,257	Application Framework
Magento	1.8.1.0	5.2.13	12/12/13	8,086	632,924	Online Retail
MediaWiki	1.22.0	5.3.2	12/6/13	1,869	1,037,124	Wiki
Moodle	2.6	5.3.3	11/18/13	6,553	852,075	Online Learning
osCommerce	2.3.3.4	4.0.0	9/26/13	569	46,804	Online Retail
PEAR	1.9.4	4.4.0	7/7/11	74	31,257	Component Framework
phpBB	3.0.12	4.3.3	9/28/13	270	149,361	Bulletin Board
phpMyAdmin	4.1.3	5.3.0	12/31/13	455	138,842	Database Administration
SilverStripe	3.1.2	5.3.2	10/22/13	572	92,216	CMS
Smarty	3.1.16	5.2.0	12/17/13	126	15,904	Template Engine
Squirrel Mail	1.4.22	4.1.0	7/12/11	276	36,082	Webmail
Symfony	2.4.0	5.3.3	12/3/13	4,023	253,536	Application Framework
WordPress	3.8.1	5.2.4	1/23/14	482	132,877	Blog
The Zend Framework	1.12.3	5.2.11	3/13/13	4,546	573,813	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 20 systems consisting of 32,682 files with 4,593,476 total lines of source.

Table 2: The PHP Corpus.

type in the corpus. We used popularity, instead of actual number of downloads or installed sites, since we have no way to accurately compute these figures. In total, the corpus consists of 32,682 PHP source files with 4,593,476 lines of PHP source (counted using the cloc[9] tool). The systems in this corpus were used during development of the includes resolution analysis, both as a test-bed, to check the results of the analysis, and as a source of usage patterns that a realistic analysis would need to handle.

5.2 Evaluating FLRES

FLRES is evaluated here based on the following questions: (a) "How many includes in real PHP code can be resolved to a small set of target files?", (b) "How small are these sets?" and (c) "How fast does the analysis usually run?". Smaller target file sets mean the algorithm is more precise, while a faster run time means it is suitable for use in an IDE.

In terms of recall and precision, we know the algorithm to have 100% recall since it is sound under assumptions that hold within the corpus. Precision could be defined per resolved include as $1/numberOfPossibleFiles$, under the assumption that only one specific file would be included at run-time. However, such assumption is not reasonable since some includes could indeed be designed to resolve to several files. We therefore simply report the number of computed candidates for each include.

5.2.1 Method

To evaluate the effectiveness of the FLRES algorithm, we have applied it to all the PHP files in the corpus shown in Table 2, using a Rascal script to run the analysis and allow for reproducibility. We use Rascal to gather the information seen in Table 3. All `include` expressions are matched just by matching any `include` expression node in the AST, while dynamic includes are defined, for matching purposes, as any include expression with an operand other than a string literal (strings built using interpolation appear as so-called *encapsed* strings instead of string literals).

Per PHP system in the corpus, we measure how many include statements there are, how many of those have paths built dynamically at runtime, which of the includes we can resolve to a single unique file, to how many candidates an include points after resolution, and how many can not be resolved at all. A strong reduction of provided includes (especially dynamic includes) to unique files, or at least a low amount of left-over candidates, indicates success. Contrary, if FLRES would not work, this method would show insignificant amounts of resolved includes or high average left-over non-determinism. Intuitively, we expected the larger part of includes to be resolved by FLRES, i.e. $> 80\%$.

5.2.2 Threats to Validity

We note the following threats to validity in our evaluation of FLRES:

1. We focused mainly on large, well-maintained software systems. It may be the case that these systems are more careful, and consistent, in how they use includes than other systems. However, we do not believe that this is an issue: uses of dynamic includes vary widely in the systems we have looked at, and the corpus contains a broad selection of software from many different

[9] `http://cloc.sourceforge.net`

System	Includes			Results				
	Total	Static	Dynamic	Unique	Missing	Any	Other	Average
CakePHP	125	4	121	55	3	22	45	4.87
CodeIgniter	69	0	69	25	13	27	4	11.00
DoctrineORM	74	2	72	55	1	18	0	0.00
Drupal	173	1	172	132	5	33	3	3.67
Gallery	47	5	42	29	2	14	2	2.50
Joomla	444	4	440	228	10	162	44	10.84
Kohana	51	4	47	6	1	41	3	2.00
Magento	193	129	64	123	2	48	20	2.60
MediaWiki	514	43	471	480	7	25	2	10.50
Moodle	8,619	3,438	5,181	6,798	114	237	1,470	138.27
osCommerce	705	149	556	90	1	41	573	2.60
PEAR	211	200	11	147	0	11	53	2.00
phpBB	415	0	415	0	0	415	0	0.00
phpMyAdmin	887	731	156	842	3	34	8	46.88
SilverStripe	554	482	72	521	8	23	2	5.00
Smarty	37	2	35	27	0	10	0	0.00
SquirrelMail	427	4	423	412	5	9	1	17.00
Symfony	246	5	241	157	16	64	9	2.33
WordPress	656	3	653	609	10	28	9	5.78
ZendFramework	13,772	13,354	418	13,523	42	67	140	2.19

Table 3: Results of running FLRES on the corpus.

domains.

2. We do not try to account for cases where part of one system is used in another, which could skew our results. This happens in `Magento`, for instance, which uses part of the `Zend Framework` libraries, and in `Gallery`, which uses part of `Kohana`. Although this could mean we are counting the same resolved cases multiple times, it could also mean we have the same unresolved cases appearing in multiple locations. We have not attempted to determine which versions of these libraries appear embedded in other projects, or whether the included code is the original code or has been modified.

3. If a system violates the soundness assumptions given in Section 3 FLRES would return incorrect results. We believe this is unlikely: it should be possible to determine which external libraries are used by a system, and we have not seen the situation we described with defined constants in any of the code we have examined.

5.2.3 *Results for* FLRES

Table 3 shows the result of running FLRES on the corpus shown in Table 2. The first column shows the name of the system. The second, third, and fourth columns provide information about the includes in this system: Total gives the total number of include expressions, Static gives the number of these includes where the file to include is given as a string literal, and Dynamic gives the number of includes that use expressions other than a string literal to specify the included file. This is a good proxy for which includes could include multiple files, but not perfect: Figure 3 illustrated that file paths given as string literals may refer to multiple files.

The five columns under Results then show the actual results of the analysis. Unique shows the total number of includes that can be assigned a single possible target file by FLRES. Missing then shows the number of includes with no possible target file. While in some cases this appears to be an error in the code, in many the missing includes are surrounded by a check to see if the file is present, and appear to be for included files which are part of optional extensions to the system. Column Any illustrates the other extreme, cases where the include could refer to 90 percent or more of the files in the system. Other then shows includes between these two extremes—includes that could refer to more than one file, but are specific enough to not refer to at least 90 percent of the files. Finally, the Average column indicates how many files, on average, each of these Other includes could actually refer to. For instance, for CakePHP, each of the includes classified as Other could refer to roughly 5 (4.87) files.

Figure 4 shows an overview of the running times per file for all files in the corpus. The plot shows that although there exist some outliers above 5 seconds (the largest outlier, at 138 seconds, is not included in this plot), and quite a few outliers that may take up to half a second, FLRES is able to analyze most of the files within 5 to 50 milliseconds, with a median of just over 5.

5.2.4 *Analysis*

In many of the systems the number of unique includes is quite high, while the average number of possible files for those includes in the Other category are much lower than the total number of files in the system, with many systems having a range from roughly 2 (Kohana, Pear, the Zend Framework, Symfony) up to around 5. As indicated above,

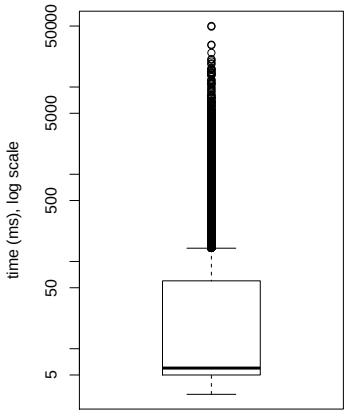

Figure 4: Boxplot detailing run-time in ms of FLRES for all files in the corpus.

performance is also good. While we are investigating the outliers to further improve performance, most files can be analyzed in 5 to 50 ms.

There are several reasons that an include may not evaluate to a Unique include, or may even evaluate to Any. First, as was shown in Figure 3, in some cases the file possibly included by the given include expression isn't unique, at least when the context isn't known. Second, in some cases the include relies on a constant which is given different values in different files. The analysis then has to take account of the inclusion relation between files to ensure the correct constant is used. The PGRES algorithm takes account of both of these limitations in FLRES, allowing it to improve the precision of the analysis and give fewer possible files per include. Third, in some cases a stronger analysis, such as a string analysis, is needed. This is the case in phpBB, which uses variables in every include expression. Finally, some of the includes are specifically designed to support plugin architectures, and use very general include expressions to facilitate this. It may be possible for the user to annotate the expression somehow to provide additional information, but in general it is not possible for any static analysis to determine the possible files to be included in this scenario.

5.3 Evaluating PGRES

PGRES is evaluated focusing on the following question: (a) "How much improvement does the algorithm produce in the size of the sets of candidates over FLRES?". Again, smaller candidate sets indicate a more precise algorithm, and can also improve the precision of other analyses using the results of PGRES. The definitions of recall and precision used here are the same as for FLRES, given above.

5.3.1 Method

The method used to evaluate PGRES is similar to that used for FLRES, except for the choice of files to analyze. Since PGRES is targeted at PHP programs, only files that could be directly invoked have been chosen. We have limited our efforts to programs from the following systems: osCommerce, WordPress, MediaWiki, phpMyAdmin, and CakePHP.

5.3.2 Threats to validity

Our evaluation of PGRES has the same threats to validity as FLRES, described above, but with the soundness assumptions for PGRES described in Section 4. Most of these soundness assumptions (e.g., setting the include path inside `eval`) are very unlikely—in this case, `eval` is very rarely used, and we have not seen this scenario in real code.

A threat unique to PGRES is that it may be the case that what we identify as a program is actually not, but is instead an include used in other files. Identifying which files can be directly invoked, versus which can be included in other files, is not trivial—the most obvious solution, to use files that cannot be included by other files, would assume the analysis we are implementing. Instead, this requires a deep knowledge of each individual system, one of the reasons we have restricted the number of systems evaluated using PGRES. In the case where we have inadvertently selected an include file, PGRES may resolve includes incorrectly, either missing target files that should be present or giving incorrect target files. Since PGRES uses the results from FLRES as a starting point, PGRES will never return a set of candidates larger than FLRES. When applying PGRES as a first analysis step enabling another, say static taint analysis, we propose that the top level files would be given as a manual configuration parameter.

5.3.3 Results

PGRES was executed on a total of 408 programs: 137 from MediaWiki, 91 from WordPress, 90 from phpMyAdmin, 88 from osCommerce, and 2 from CakePHP (which is a framework, but includes a small sample application). PGRES was not able to improve upon the results of FLRES for either MediaWiki or WordPress, but improvements were seen with the other three applications. The numbers below are computed by looking at all includes identified as being reachable by PGRES for a specific program, then, for each include, computing the number of candidates returned by both FLRES and PGRES. PGRES is more precise in those cases where more `include` expressions have smaller candidate sets.

- In the case of phpMyAdmin, summing across all programs analyzed, PGRES was able to reduce the total number of includes categorized as Any from 188 to 89. PGRES was also able to reduce the number with a candidate set of 4 from 178 to 89, and of 2 from 267 to 178, while increasing the number with a unique match from 73,425 to 73,692.
- In the case of CakePHP, across the 2 programs analyzed, PGRES was able to reduce the total number of includes with a candidate set of 6 from 20 to 0, of 4 from 2 to 0, and of 2 from 10 to 6, while increasing the number with a unique match from 108 to 134.
- In the case of osCommerce, summing across all programs analyzed, PGRES was able to reduce the total number of includes with a candidate set of 10 from 1232 to 0; of 9 from 704 to 0; of 8 from 176 to 0; of 7 from 88 to 0; of 6 from 1496 to 88; of 5 from 616 to 88; of 4 from 440 to 264; of 3 from 1848 to 88; and of 2 from 42,944 to 8,100. The number of includes with a unique match went from 7,832 to 48,748.

The median execution time of PGRES over the 408 analyzed programs was 17.483 seconds and the average was 20.962 seconds, with some programs analyzed almost immediately and one taking 52.209 seconds.[10]

[10] A more detailed performance analysis will be included in

5.3.4 Analysis

The results of the analysis show that PGRES is quite effective specifically in those cases where contextual information about the script is important, especially when the directory of the including script or the values of reachable (versus globally uniquely defined) constants determine which files can be imported. In situations where this is not the case, PGRES does not improve on the results computed by FLRES. In both MediaWiki and WordPress, the includes that cannot be further resolved are generally to support system extensions (e.g., the final example in Figure 2, which supports installable "skins" that can be selected by the user as part of his/her preferences), with file names given in terms of global variables, method parameters, method results, and object fields, based on data from the site configuration or the database. In some cases a stronger analysis might be able to further reduce the target file sets, but one lesson learned from this evaluation is that the chance of success may be low. Since PGRES is slower than FLRES, it seems not suitable for interactive use in an IDE setting. Instead PGRES would be applied as part of other program analysis tools which require higher accuracy.

6. RELATED WORK

The inability to resolve includes has been a challenge for creating realistic PHP analysis tools. For instance, Huang et al's WebSSARI system [7] does not handle dynamic includes at all, leading to a need to manually "resolve" the includes and manually merge the included code. Jovanovic et al.'s Pixy [9, 8] is able to resolve some dynamic includes, but this resolution mechanism appears to fail in practice [2]. The analysis used by Pixy for resolving dynamic includes appears to not perform path matching as is done in our algorithm, and also appears to assume the values of constants are known in advance, instead of being dependent on which definition of each constant is actually reachable based on file inclusion. Finally, since Pixy only supports PHP 4, it does not handle new PHP features, such as class constants, that can be used in include path expressions.

Wassermann et al. [16, 17], extending earlier work by Minamide [12], model string values and string operations in PHP programs using context-free grammars and language transducers, respectively. Although their analysis was focused on preventing injection attacks and cross-site scripting vulnerabilities, they apparently also used their string analysis to resolve dynamic includes, including using a path matching technique which seems similar to ours. Unfortunately their tool appears to no longer be available, making it challenging to compare their results with ours.

Biggar [1], in his work on the `phc` compiler, looked at a large corpus of 581 PHP code packages downloaded from SourceForge, totaling 42,523 files with 8,130,837 lines of code (including blank lines and comments). He found fewer dynamic includes in his dataset than we have in ours—only 15% of the includes in his corpus are dynamic. This may be just a difference of classification—we consider any include to be dynamic that has a non-literal operand, while Biggar also considers include expressions built using string literals, concatenation, and constants to be static (we do not for the reason mentioned above—constant definitions are not globally unique through the entire system). Zhao et al. [18], in their work on the Facebook HipHop compiler, instead impose a requirement that all includes are statically known at compile time, eliminating many of the legitimate uses we have found for this technique (for instance, to support installable plugins). The algorithm presented here would loosen this requirement to allow includes that are static in practice, while still allowing the flexibility of building the file name to be included using common string-building operations and reachable constant definitions.

Looking at other dynamic scripting languages with similar features, Richards et al. [15] 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. [14] 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. [11] then used these results to create a tool, Evalorizer, that can be used to remove many uses of `eval` found in JavaScript code. Although aimed at eliminating uses of `eval`, not dynamic includes, this work has a goal similar to ours.

This is also the case with the work of Furr et al. [4], who 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 [3]. A similar approach was taken by Mulder [13] for PHP, where profiling information was used to replace occurrences of `call_user_func` and `call_user_func_array`, used to dynamically call functions and methods by name, with actual calls to these functions and methods.

7. CONCLUSIONS

PHP's `include` expressions are an example of a dynamic feature that has traditionally made static analysis of PHP challenging. In this work we presented two algorithms, FLRES and PGRES, for resolving `include` expressions at the level of individual files and PHP programs, respectively. Evaluating FLRES over a corpus of more than 4.5 million lines of PHP code and more than 32,000 files, we showed that FLRES is effective at producing small sets of candidate files for many `include` expressions, and generally runs fast enough to be used in PHP IDEs.

PGRES, evaluated over more than 400 PHP programs—scripts designed to be executed directly—shows significant improvements on the results of FLRES in some cases, with little to no improvement in others. PGRES can be applied when accuracy is more important than efficiency, for example when analyzing the security of PHP programs.

In future work we plan to make use of both algorithms in our ongoing work on the PHP AiR framework for PHP program analysis. Finally, we plan to continue ongoing work on integrating PHP analysis with the Eclipse IDE, with the goal of providing IDE support to support developers in developing, understanding, and refactoring large PHP code bases.

the final version.

8. REFERENCES

[1] P. Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin, April 2010.

[2] N. L. de Poel. Automated Security Review of PHP Web Applications with Static Code Analysis. Master's thesis, University of Groningen, 2010.

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

[4] 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.

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

[6] M. Hills, P. Klint, and J. J. Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of ISSTA'13*, pages 325–335. ACM, 2013.

[7] 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*, pages 40–52. ACM, 2004.

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

[9] 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*, pages 258–263, 2006.

[10] 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.

[11] 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.

[12] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of WWW 2005*, pages 432–441. ACM, 2005.

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

[14] 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.

[15] 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.

[16] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of PLDI'07*, pages 32–41. ACM, 2007.

[17] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of ICSE'08*, pages 171–180. ACM, 2008.

[18] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The HipHop Compiler for PHP. In *Proceedings of OOPSLA'12*, pages 575–586. ACM, 2012.