An Empirical Study of PHP Feature Usage: A Static Analysis Perspective

Mark Hills, Paul Klint, and Jurgen J. Vinju
CWI, Software Analysis and Transformation (SWAT)

ISSTA 2013
Lugano, Switzerland
July 16-18, 2013

http://www.rascal-mpl.org

PHP


Create account
Log in
WIKIPEDIA
The Free Encyclopedia
Article
Talk
Read
Edit
View history
Search
PHP
From Wikipedia, the free encyclopedia
(Redirected from Php)
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Interaction
Help
About Wikipedia
This article is about the scripting language. For other uses, see PHP (disambiguation).
PHP is a server-side scripting language designed for web development but also used as a general-purpose programming language. PHP is now installed on more than 244 million websites and 2.1 million web servers.[2] Originally created by Rasmus Lerdorf in 1995, the reference implementation of PHP is now produced by The PHP Group.[3] While PHP originally stood for Personal Home Page,[4] it now stands for PHP: Hypertext Preprocessor, a recursive acronym.[5]
PHP code is interpreted by a web server with a PHP processor module which generates the resulting web page: PHP commands can be embedded directly into an HTML source
PHP
php
phpmaster.com
What's New in PHP 5.5
4
| By: Patrick Mulvey | Posted: July 12, 2013 | News & Opinion

PHP Analysis in Rascal (PHP AiR)

- Big picture: develop a framework for PHP source code analysis
- Domains:
  - Program analysis (static/dynamic)
  - Software metrics
  - Empirical software engineering
  - Developer tool support

Why look at PHP applications?

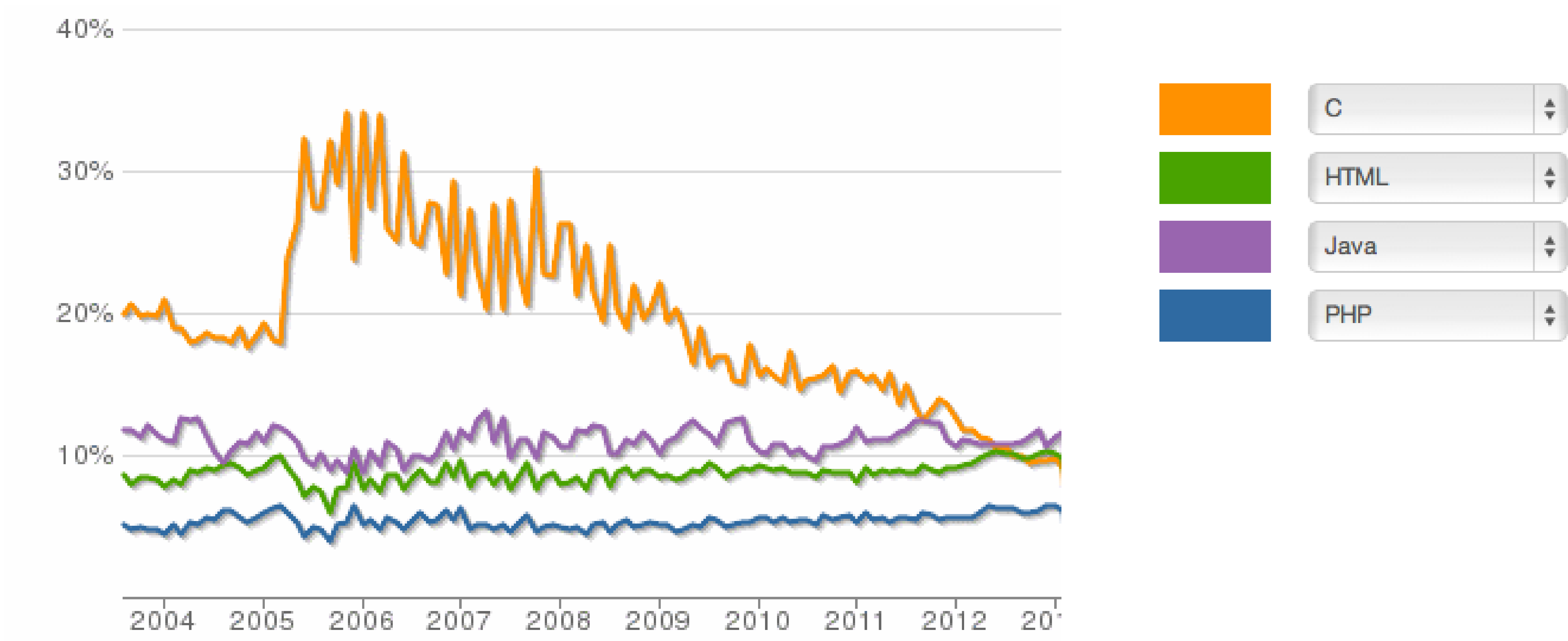
Why look at PHP applications?

Why look at PHP applications?

Why look at PHP applications?

PHP applications are everywhere!

Open Source Commits by Language (Ohloh.net)



http://www.ohloh.net/languages/compare?measure=commits&percent=true

Challenges in Tool Development

Example: Building a type inferencer

Example: Building a type inferencer

- Lots of different statements and expressions, are they all used? What do we need to implement first to get up and going?

Example: Building a type inferencer

- Lots of different statements and expressions, are they all used? What do we need to implement first to get up and going?

- What if the code has evals? This could add new types.

Example: Building a type inferencer

- Lots of different statements and expressions, are they all used? What do we need to implement first to get up and going?
- What if the code has evals? This could add new types.
- What if the code has invocation functions? Can we tell what functions are called?

Example: Building a type inferencer

- Lots of different statements and expressions, are they all used? What do we need to implement first to get up and going?
- What if the code has evals? This could add new types.
- What if the code has invocation functions? Can we tell what functions are called?
- What if the code contains variable variables? Can we tell which variables they refer to?

Example: Building a type inferencer

- Lots of different statements and expressions, are they all used? What do we need to implement first to get up and going?
- What if the code has evals? This could add new types.
- What if the code has invocation functions? Can we tell what functions are called?
- What if the code contains variable variables? Can we tell which variables they refer to?
- What if...

Looking more generally

- PHP is big, which language features should we focus on first?
- PHP is dynamic, how much impact do these features have on real programs?
- What kinds of assumptions (e.g., no evals, no writes through variable variables) can we safely make about code and still have good precision?
- How can we build prototypes that work with real PHP code?

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1, 105–133 (1971)

An Empirical Study of FORTRAN Programs†

DONALD E. KNUTH

Computer Science Department, Stanford University, Stanford, California 94305

SUMMARY

A sample of programs, written in FORTRAN by a wide variety of people for a wide variety of applications, was chosen 'at random' in an attempt to discover quantitatively 'what programmers really do'. Statistical results of this survey are presented here, together with some of their apparent implications for future work in compiler design. The principal conclusion which may be drawn is the importance of a program 'profile', namely a table of frequency counts which record how often each statement is performed in a typical run; there are strong indications that profile-keeping should become a standard practice in all computer systems, for casual users as well as system programmers. This paper is the report of a three month study undertaken by the author and about a dozen students and representatives of the software industry during the summer of 1970. It is hoped that a reader who studies this report will obtain a fairly clear conception of how FORTRAN is being used, and what compilers can do about it.

KEY WORDS FORTRAN Optimization Efficiency Compiler

Solution: Study PHP feature usage *empirically*

- What does a typical PHP program (level of focus: individual pages) look like?

- What features of PHP do people really use?

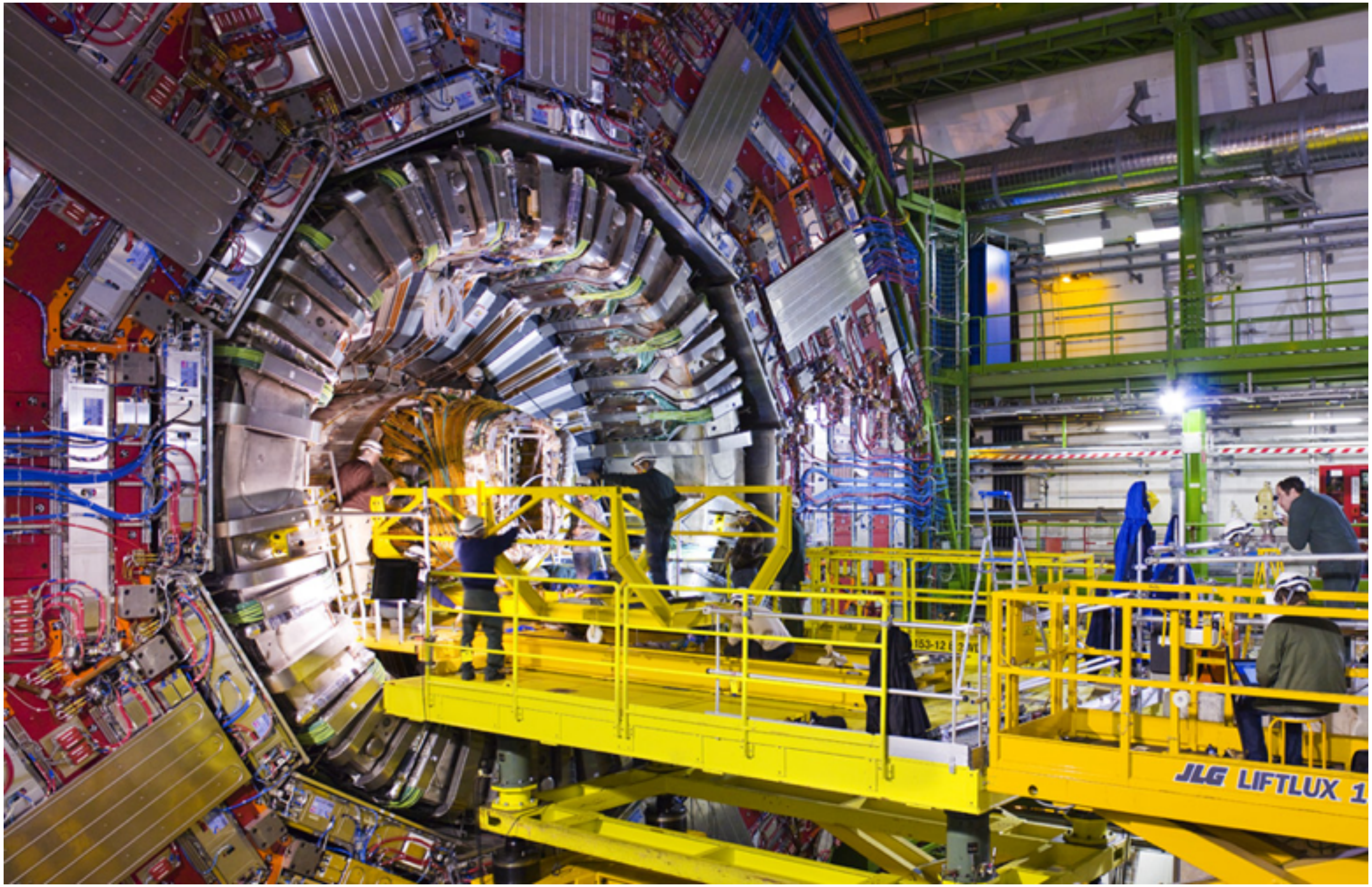
- How often are dynamic features, which are hard for static analysis to handle, used in real programs?

- When dynamic features appear, are they really dynamic? Or are they used in static ways?

Which dynamic features?

- **Dynamic includes**
- **Variable Constructs**
- Overloading
- **eval**
- Variadic Functions
- Dynamic Invocation

Setting Up the Experiment: Tools & Methods



http://cache.boston.com/universal/site_graphics/blogs/bigpicture/lhc_08_01/lhc11.jpg

Building an open-source PHP corpus

- Well-known systems and frameworks: WordPress, Joomla, MediaWiki, Moodle, Symfony, Zend
- Multiple domains: app frameworks, CMS, blogging, wikis, eCommerce, webmail, and others
- Selected based on Ohloh rankings, based on popularity and desire for domain diversity
- Totals: 19 open-source PHP systems, 3.37 million lines of PHP code, 19,816 files

Methodology

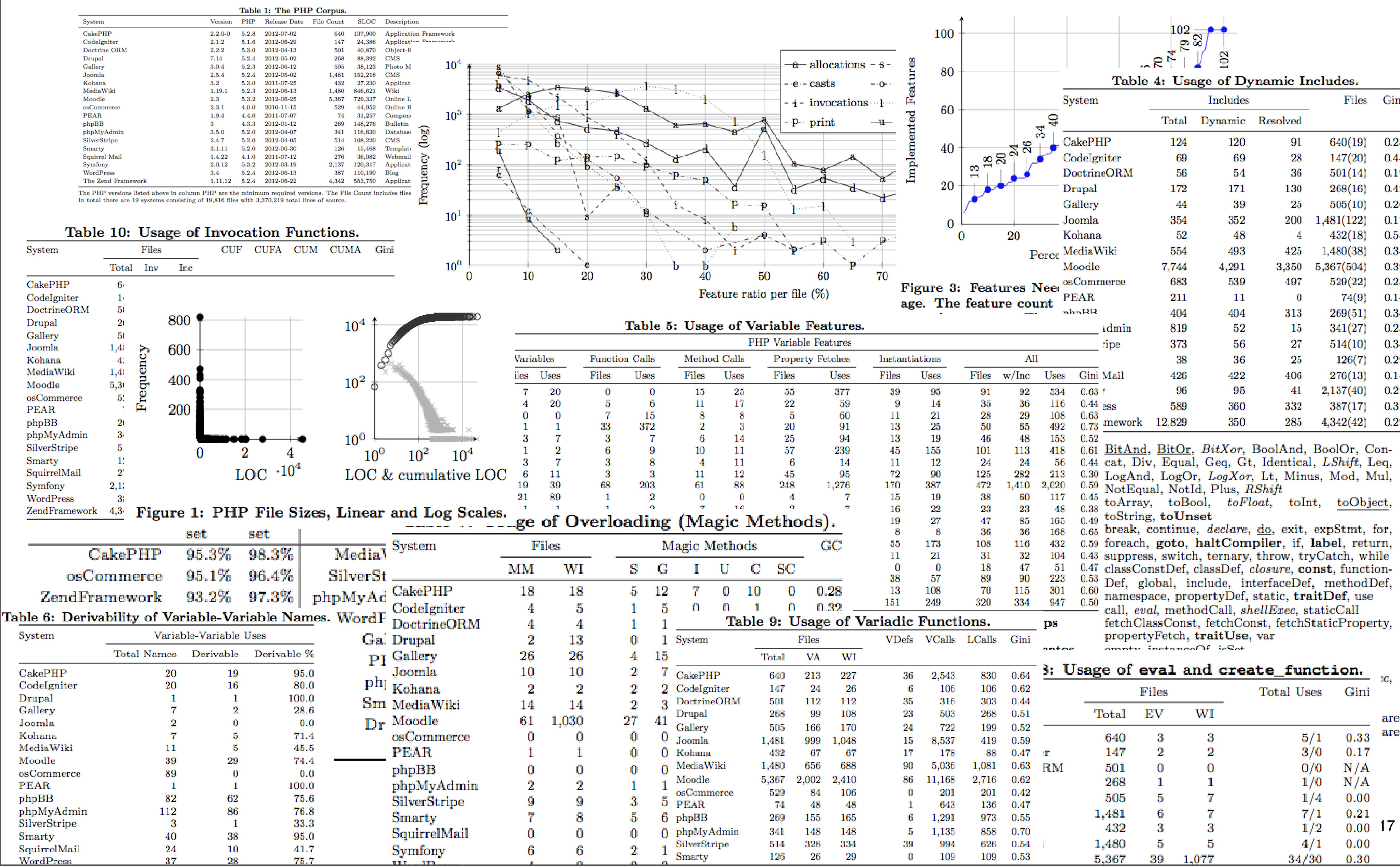
- Corpus parsed with an open-source PHP parser
- Feature usage extracted directly from ASTs
- Dynamic features identified using pattern matching
- More in-depth explorations performed manually or using custom-written analysis routines
- All computation scripted, resulting figures and tables generated

  - http://www.rascal-mpl.org/

Threats to validity

- Results could be very corpus-specific
- Large, well-known open-source PHP systems may not be representative of typical PHP code
- Dynamic includes could skew results

Interpreting the Results



Thursday, July 18, 13

Zooming in

- Feature usage and coverage
- Dynamic includes
- Variable variables
- eval

Feature usage and coverage

- Goal: analysis prototypes should cover actual programs
- Solution: compute which sets of features cover the most files
- 109 features total
  - 7 never used (including goto), mainly newer features
  - casts, predicates, unary operations used rarely
  - 74 features cover 80% of all files, over 90% for some systems (CakePHP: 95.3%, Zend: 93.2%)

Dynamic includes

```
require_once( dirname( __FILE__ ) . '/Maintenance.php' );

$maintananceDir = dirname( dirname( dirname( dirname(
        dirname( __FILE__ ) ) ) ) ) . '/maintenance';
require( "$maintananceDir/Maintenance.php" );
```

- In PHP, may not know code that will run until runtime
- Q1: How often are dynamic includes used?
- Q2: How often can we resolve them to a specific file up front?

Usage of dynamic includes

- 19,816 files in corpus: 3,184 contain dynamic includes (16.1%)
- 25,637 includes in corpus: 7,962 are dynamic (31.1%)
- Some systems worse than others: CakePHP (120 of 124 includes are dynamic), CodeIgniter (69 of 69), Drupal (171 of 172), Moodle (4291 of 7744)
- Some only use in limited way: Zend only 350 of 12,829 are dynamic, PEAR only 11 of 211

Resolution of dynamic includes

- After resolution, 864 files contain dynamic includes (27.1% of files with dynamic includes still contain them, 4.4% of total files)
- After resolution, 1,439 dynamic includes remain (18.2% of original)
- Based on current resolution analysis, dynamic includes usually not brought in through other includes
- Results on major systems: Drupal (130 of 171 resolved), Joomla (200 of 352 resolved), MediaWiki (425 of 493), Moodle (3350 of 4291), WordPress (332 of 360), Zend (285 of 350)
- Not always so good: 4 of 48 in Kohana resolved, 41 of 95 in Symfony, 0 of 11 in PEAR

Variable variables

```
$x = 3;
$y = 'x';
echo $x; // 3
echo $y; // x
echo $$y; // 3
$$y = 4;
echo $x; // 4
```

- Reflective ability to refer to variables using strings
- Often used as a code saving device
- Problem: creates aliases using string operations

Variable variables: findings

- Question: How often can we statically determine to which names a variable variable can refer?
- Method: use Rascal to find all locations of variable variables, manually inspect code
- Restrictions: names statically determinable, no aliases, no other declarations
- General: 61% of uses resolvable, 75% in newer systems
- Best: 100% in Drupal & PEAR, 95% in CodeIgniter & Smarty
- Worst: 0% in Joomla & osCommerce

The eval expression (and create_function)

```
eval(str_replace(array('<?php', '?>'), '', $result['code']));

create_function('$v',
  '$v[\'title\'] = $v[\'title\'] . \'-transformed\'; return $v;')
```

- eval and create_function provide for runtime evaluation of arbitrary code
- Used rarely in corpus: 148 occurrences of eval, 72 of create_function, many uses in testing and maintenance code
- Uses truly dynamic, need string analysis and (in the general case) dynamic analysis to determine actually invoked code

Occurrences of all dynamic features

- 19,816 files in corpus: 3,386 contain dynamic features (17.1%)
- Dynamic feature usage varies greatly over systems
    - PEAR: 50% of files have at least 1 dynamic feature
    - WordPress: 30.7%
    - MediaWiki: 14.6%
    - Symfony: 9.4%

Summary

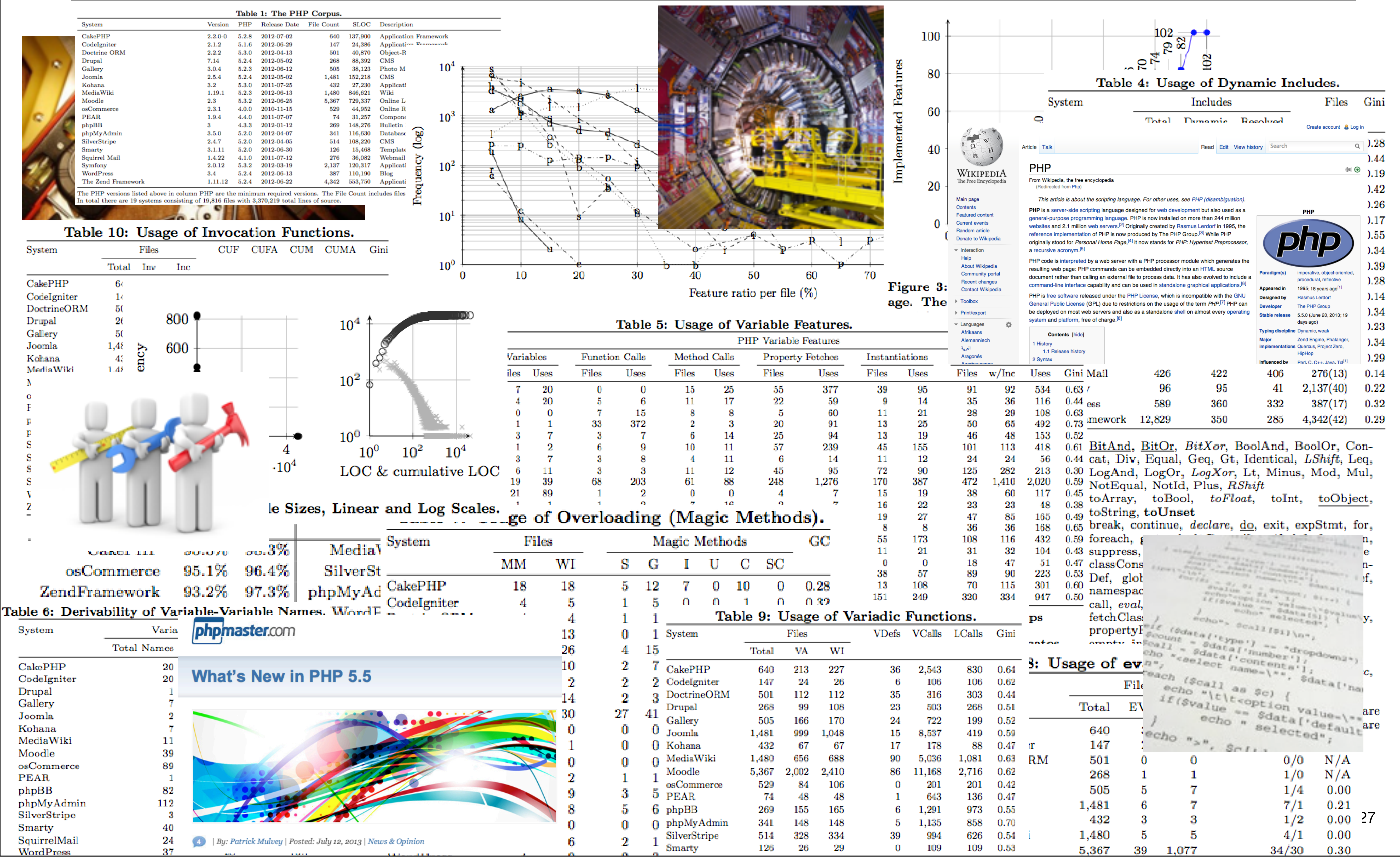


Table 1: The PHP Corpus.
Table 10: Usage of Invocation Functions.
Frequency (log)
Feature ratio per file (%)
Implemented Features
Table 4: Usage of Dynamic Includes.
WIKIPEDIA
The Free Encyclopedia
PHP
LOC & cumulative LOC
Table 5: Usage of Variable Features.
Table 6: Derivability of Variable-Variable Names.
Table 9: Usage of Variadic Functions.
phpmaster.com
What's New in PHP 5.5

Summary: What have we learned?

- Prototypes can be built to cover a subset of the language and still cover a significant number of real program files

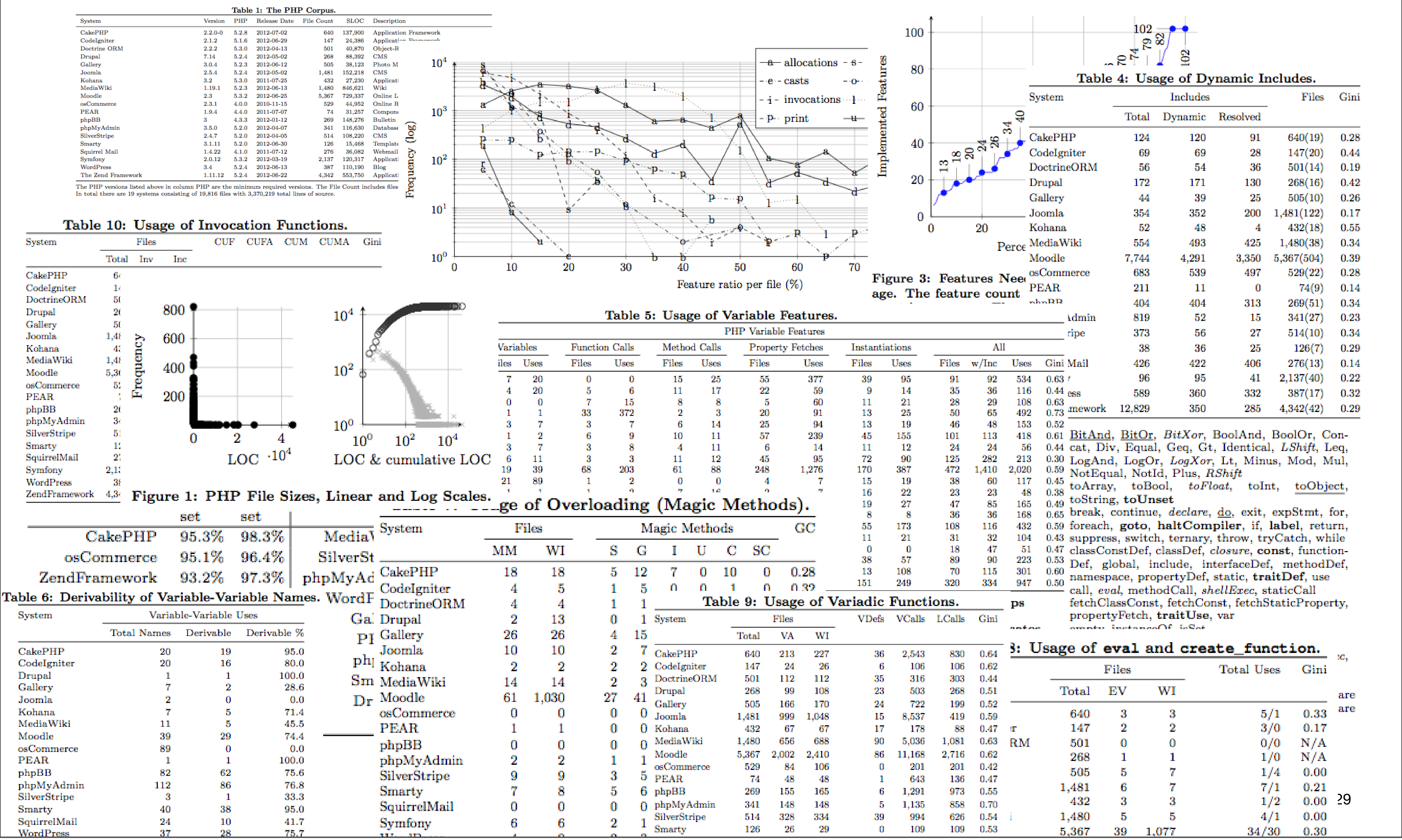
- Knowledge of how often dynamic features appear provides firmer ground for assumptions we make in building analyses

- Patterns of dynamic feature usage can be exploited in analysis tools to improve precision, mitigate against dynamic effects

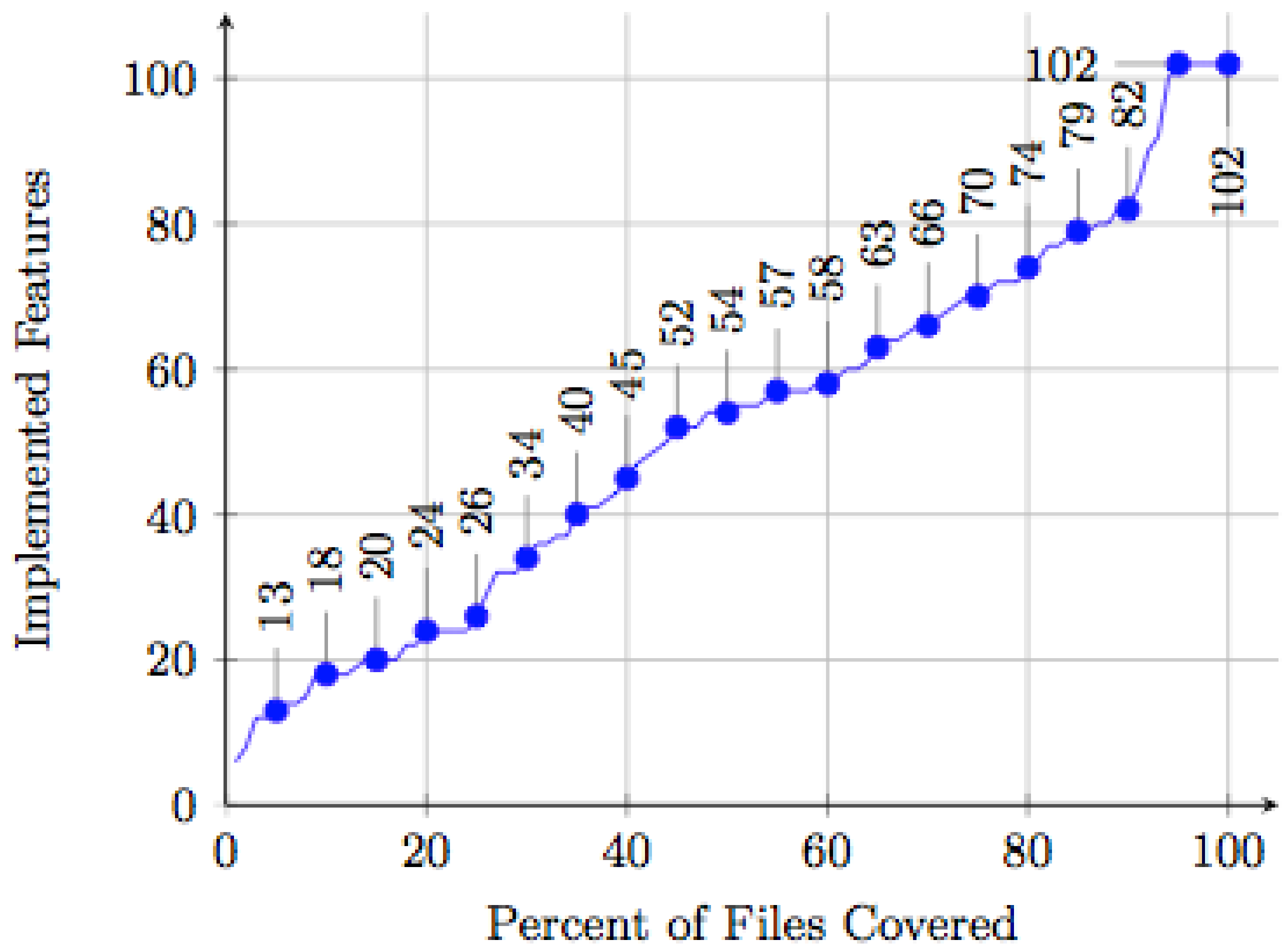
- Need to look more closely at how PHP files are used (e.g., user facing vs. unit test code), application phases (e.g., plugin initialization), may be able to leverage this

- Hybrid static/dynamic solutions are clearly needed in some cases

Backup Slides



System Feature Coverage: Overall



Related Work in JavaScript

An Analysis of the Dynamic Behavior of JavaScript Programs

Gregor Richards Sylvain Lebresne Brian Burg Jan Vitek
S3 Lab, Department of Computer Science, Purdue University, West Lafayette, IN
{gkrichar,slebresn,bburg,jv}@cs.purdue.edu

Abstract

The JavaScript programming language is widely used for web programming and, increasingly, for general purpose computing. As such, improving the correctness, security and performance of JavaScript applications has been the driving force for research in type systems, static analysis and compiler techniques for this language. Many of these techniques aim to reign in some of the most dynamic features of the language, yet little seems to be known about how programmers actually utilize the language or these features. In this paper we perform an empirical study of the dynamic behavior of a corpus of widely-used JavaScript programs, and analyze how and why the dynamic features are used. We report on the degree of dynamism that is exhibited by these JavaScript programs and compare that with assumptions commonly made in the literature and accepted industry benchmark suites.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Experimentation, Languages, Measurement

Keywords Dynamic Behavior, Execution Tracing, Dynamic Metrics, Program Analysis, JavaScript

becoming a general purpose computing platform with office applications, browsers and development environments [15] being developed in JavaScript. It has been dubbed the "assembly language" of the Internet and is targeted by code generators from the likes of Java[2,3] and Scheme [20]. In response to this success, JavaScript has started to garner academic attention and respect. Researchers have focused on three main problems: security, correctness and performance. Security is arguably JavaScript's most pressing problem: a number of attacks have been discovered that exploit the language's dynamism (mostly the ability to access and modify shared objects and to inject code via `eval`). Researchers have proposed approaches that marry static analysis and runtime monitoring to prevent a subset of known attacks [6, 12, 21, 27, 26]. Another strand of research has tried to investigate how to provide better tools for developers for catching errors early. Being a weakly typed language with no type declarations and only run-time checking of calls and field accesses, it is natural to try to provide a static type system for JavaScript [2, 1, 3, 24, 13]. Finally, after many years of neglect, modern implementations of JavaScript have started to appear which use state of the art just-in-time compilation techniques [10].

In comparison to other mainstream object-oriented languages, JavaScript stakes a rather extreme position in the spectrum of dynamicity. Everything can be modified, from the fields and methods of an object to its parents. This presents a challenge to static analy-

Proceedings of PLDI 2010, pages 1 - 12

Related Work in JavaScript

Tool-supported Refactoring for JavaScript

Asger Feldthaus *
Aarhus University
asf@cs.au.dk

Todd Millstein †
University of California, Los Angeles
todd@cs.ucla.edu

Anders Møller *
Aarhus University
amoeller@cs.au.dk

Max Schäfer
University of Oxford
max.schaefer@cs.ox.ac.uk

Frank Tip
IBM Research
ftip@us.ibm.com

Abstract

Refactoring is a popular technique for improving the structure of existing programs while maintaining their behavior. For statically typed programming languages such as Java, a wide variety of refactorings have been described, and tool support for performing refactorings and ensuring their correctness is widely available in modern IDEs. For the JavaScript programming language, however, existing refactoring tools are less mature and often unable to ensure that program behavior is preserved. Refactoring algorithms that have been developed for statically typed languages are not applicable to JavaScript because of its dynamic nature.

1. Introduction

Refactoring is the process of improving the structure of software by applying behavior-preserving program transformations [9], and has become an integral part of current software development methodologies [4]. These program transformations, themselves called refactorings, are typically identified by a name, such as RENAME FIELD, and characterized by a set of preconditions under which they are applicable and a set of algorithmic steps for transforming the program's source code. Checking these preconditions and applying the transformations manually is tedious and error-prone, so interest in automated tool support for refactorings has been

Proceedings of OOPSLA 2011, pages 119 - 137

Related Work in Ruby

Profile-Guided Static Typing for Dynamic Scripting Languages

Michael Furr Jong-hoon (David) An Jeffrey S. Foster
University of Maryland
{furr,davidan,jfoster}@cs.umd.edu

Abstract

Many popular scripting languages such as Ruby, Python, and Perl include highly dynamic language constructs, such as an eval method that evaluates a string as program text. While these constructs allow terse and expressive code, they have traditionally obstructed static analysis. In this paper we present $\mathcal{P}$Ruby, an extension to Diamondback Ruby (DRuby), a static type inference system for Ruby. $\mathcal{P}$Ruby augments DRuby with a novel dynamic analysis and transformation that allows us to precisely type uses of highly dynamic constructs. $\mathcal{P}$Ruby's analysis proceeds in three steps. First, we use run-time instrumentation to gather per-application profiles of dynamic feature usage. Next, we replace dynamic features with statically analyzable alternatives based on the profile. We also add instrumentation to safely handle cases when subsequent runs do not match the profile. Finally, we run DRuby's static type inference on the transformed code to enforce type safety.

Keywords Ruby, profile-guided analysis, RIL, Scripting Languages

1. Introduction

Many popular, object-oriented scripting languages such as Ruby, Python, and Perl are dynamically typed. Dynamic typing gives programmers great flexibility, but the lack of static typing can make it harder for "little" scripts to grow into mature, robust code bases. Recently, we have been developing Diamondback Ruby (DRuby), a tool that brings static type inference to Ruby.[1] DRuby aims to be simple enough for programmers to use while being expressive enough to precisely type typical Ruby programs. In prior work, we showed that DRuby could successfully infer types for small Ruby scripts (Furr et al. 2009c).

However, there is a major challenge in scaling up static typing to large script programs: Scripting languages typically include a range of hard-to-analyze, highly dynamic

Proceedings of OOPSLA 2009, pages 283 - 300

Other Related Work: Analysis for Dynamic Languages

"Eval Begone!: Semi-Automated Removal of eval from JavaScript Programs", Fadi Meawad, Gregor Richards, Floréal Morandat, Jan Vitek. OOPSLA 2012.

"Tool-supported Refactoring for JavaScript", Asger Feldthaus, Todd D. Millstein, Anders Møller, Max Schäfer, Frank Tip. OOPSLA 2011.

"The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications", Gregor Richards, Christian Hammer, Brian Burg, Jan Vitek. ECOOP 2011.

"Type Analysis for JavaScript", Simon Holm Jensen, Anders Møller, Peter Thiemann. SAS 2009.

Related Work: Program Analysis for PHP

"The HipHop Compiler for PHP", Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar,Jason Evans, Stephen Tu. OOPSLA 2012.

"Design and Implementation of an Ahead-of-Time Compiler for PHP", Paul Biggar. PhD Thesis, Trinity College Dublin, April 2010.

"Static Detection of Cross-Site Scripting Vulnerabilities", Gary Wassermann, Zhendong Su. ICSE 2008.

"Sound and Precise Analysis of Web Applications for Injection Vulnerabilities", Gary Wassermann, Zhendong Su. PLDI 2007.

System Feature Coverage: Per System

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

Current uses & future work

- First target: resolution of dynamic includes
- Current work: string resolution (possibly incorporating earlier work)
- Investigating hybrid static/dynamic approaches, staged analysis for plugin architectures
- Need to look at segmenting system into user-facing, developer, and admin parts, get more fine grained results

WE the PEOPLE YOUR VOICE IN OUR GOVERNMENT

WE PETITION THE OBAMA ADMINISTRATION TO:

Secure resources and funding, and begin construction of a Death Star by 2016.

Those who sign here petition the United States government to secure funding and resources, and begin construction on a Death Star by 2016.

By focusing our defense resources into a space-superiority platform and weapon system such as a Death Star, the government can spur job creation in the fields of construction, engineering, space exploration, and more, and strengthen our national defense.

https://petitions.whitehouse.gov/petition/secure-resources-and-funding-and-begin-construction-death-star-2016/wlfKzFkN