



Centrum Wiskunde & Informatica

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

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction
Help
About Wikipedia

Article Talk

Read Edit View history

Search

PHP

From Wikipedia, the free encyclopedia
(Redirected from [Php](#))

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



object-oriented,
reflective

years ago^[1]

rdorf

roup

20, 2013; 19

reak

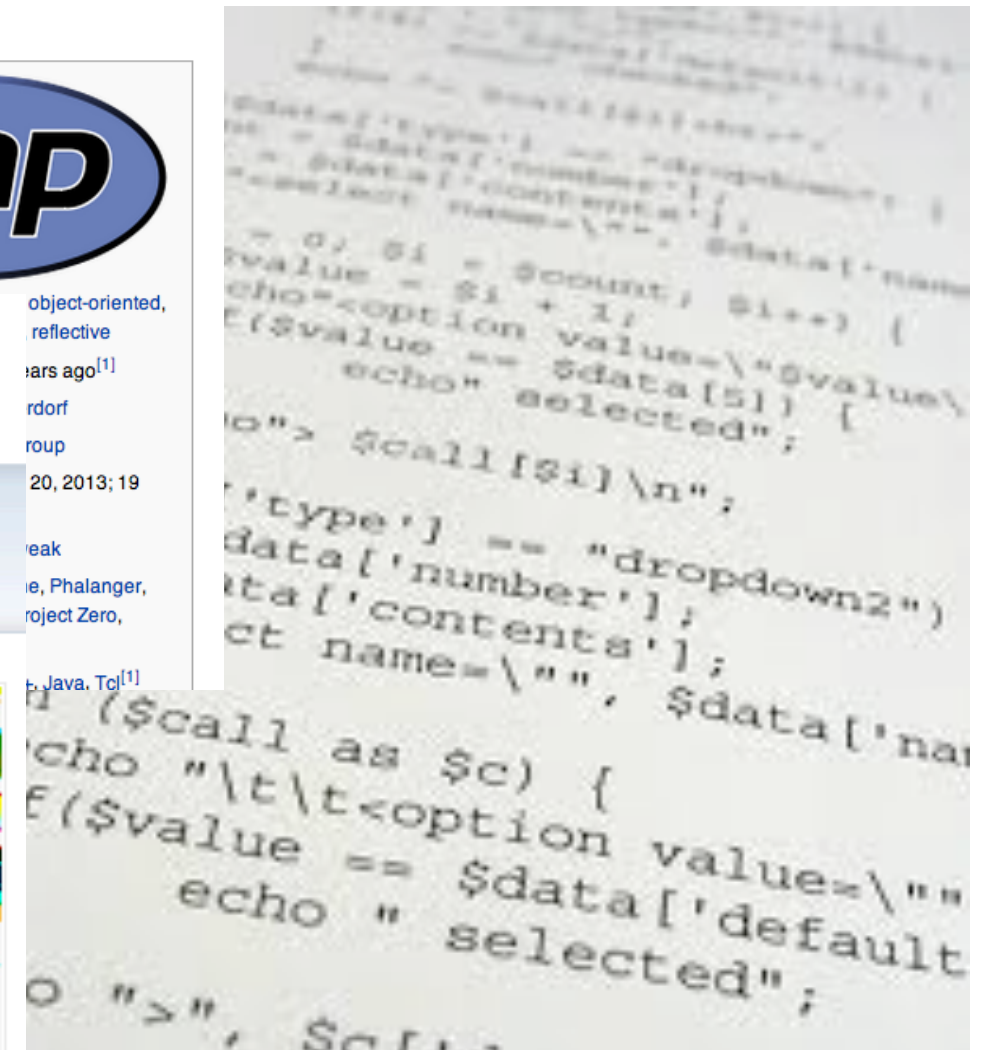
ie, Phalanger,

roject Zero,

Java. To^[1]

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?

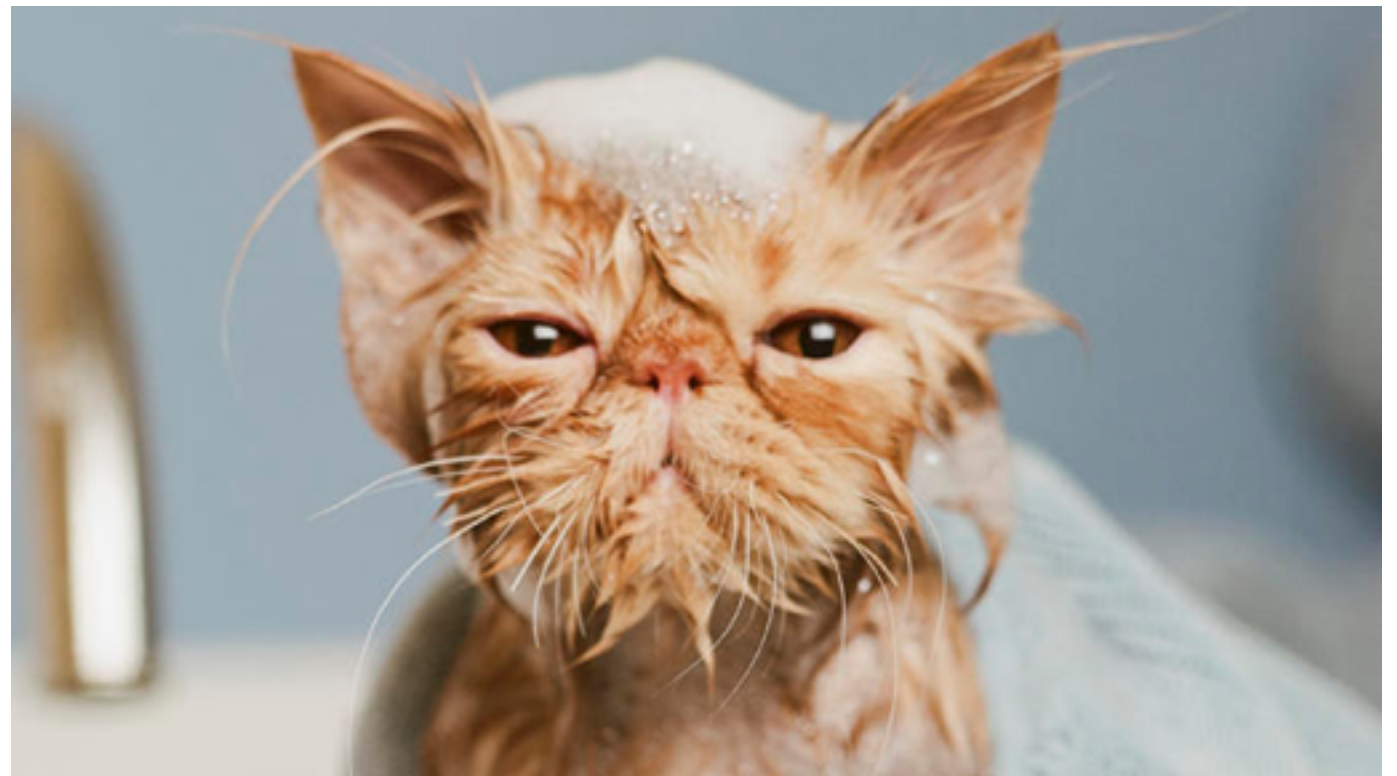


Why look at PHP applications?





Why look at PHP applications?



PHP applications are everywhere!



The New York Times

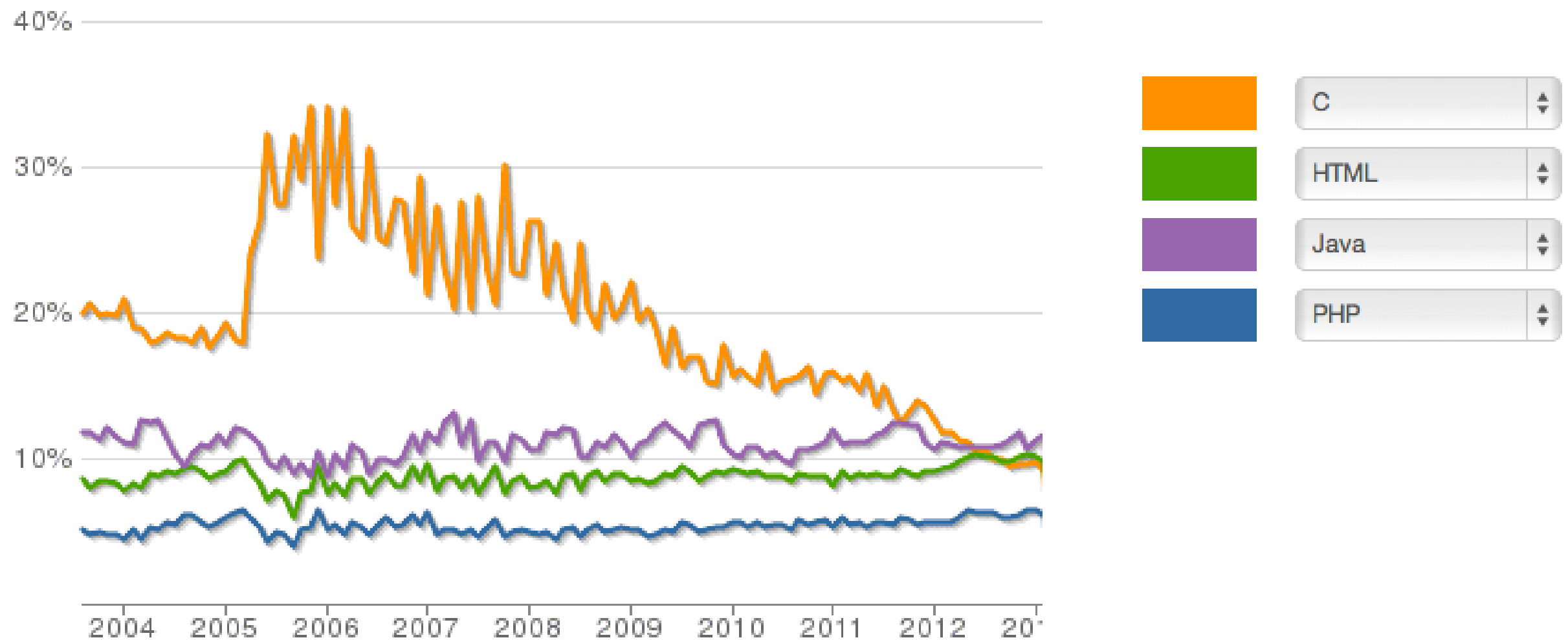


YAHOO!



SOURCEFORGE

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*

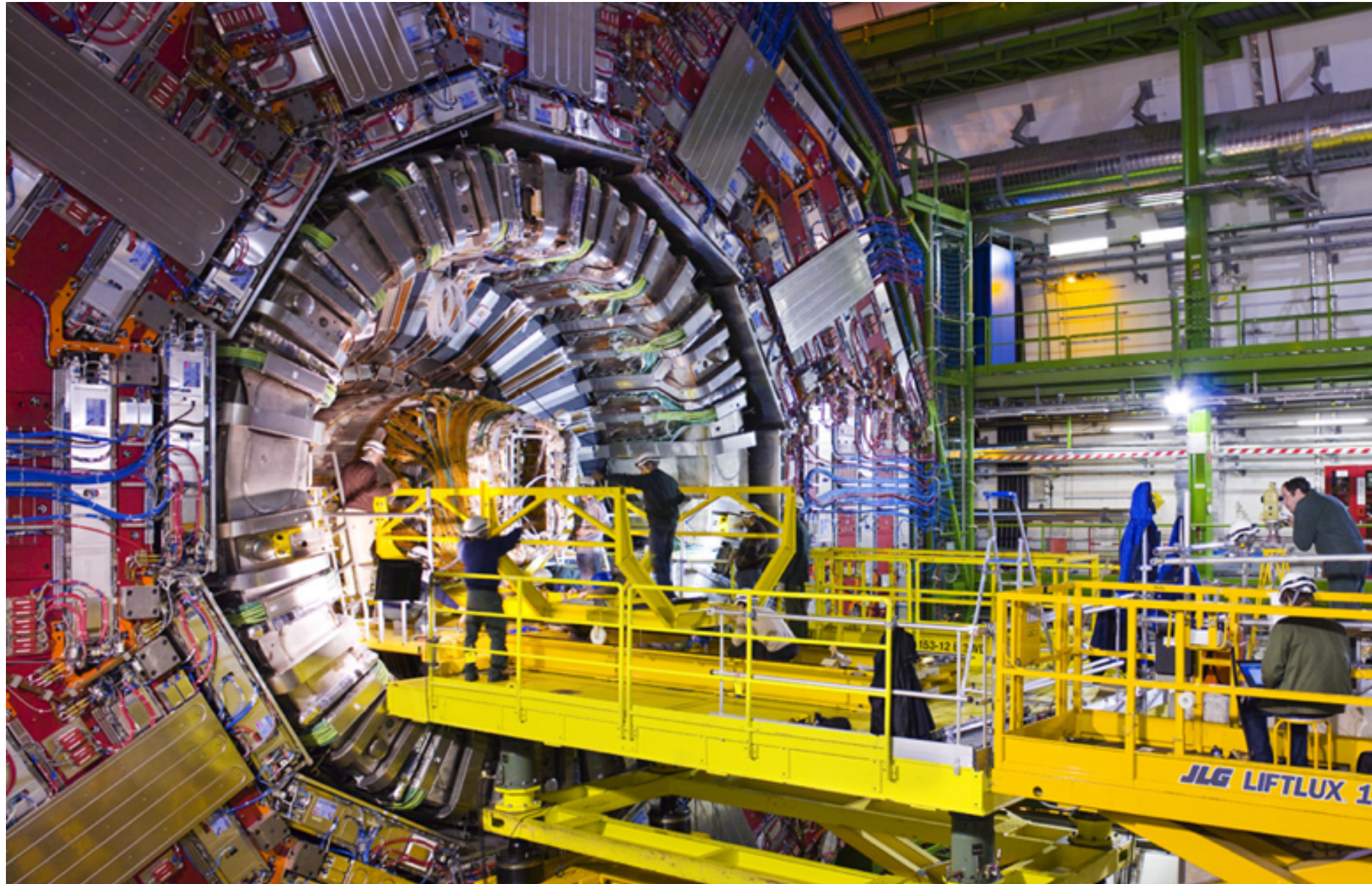
- What does a typical PHP program (level of focus: individual pages) look like?
- What features of PHP do people really use?
- How often are dynamic features, which are hard for static analysis to handle, used in real programs?
- When dynamic features appear, are they really dynamic? Or are they used 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

Table 1: The PHP Corpus.

System	Version	PHP	Release Date	File Count	SLOC	Description
CakePHP	2.2.0-0	5.2.8	2012-07-02	640	137,900	Application Framework
CodeIgniter	2.1.2	5.1.6	2012-06-29	147	24,386	Application Framework
Doctrine ORM	2.2.2	5.3.0	2012-04-13	501	40,870	Object-R
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 M
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	Applicati
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 L
osCommerce	2.3.1	4.0.0	2010-11-15	529	44,952	Online R
PEAR	1.9.4	4.4.0	2011-07-07	74	31,257	Compon
phpBB	3	4.3.3	2012-01-12	269	148,276	Bulletin
phpMyAdmin	3.5.0	5.2.0	2012-04-07	341	116,630	Database
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	Templat
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	Applicati
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	Applicati

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

Table 10: Usage of Invocation Functions.

System	Files			CUF	CUFA	CUM	CUMA	Gini
	Total	Inv	Inc					
CakePHP	6							
CodeIgniter	1							
DoctrineORM	5							
Drupal	2							
Gallery	5							
Joomla	1,4							
Kohana	4							
MediaWiki	1,4							
Moodle	5,3							
osCommerce	5							
PEAR								
phpBB	2							
phpMyAdmin	3							
SilverStripe	5							
Smarty	1							
SquirrelMail	2							
Symfony	2,1							
WordPress	3							
ZendFramework	4,3							

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

	set	set
CakePHP	95.3%	98.3%
osCommerce	95.1%	96.4%
ZendFramework	93.2%	97.3%

MediaWiki
SilverStripe
phpMyAdmin
WordPress
Gallery
PI
ph
Sm
Dr

System

Table 5: Usage of Variable Features.

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
7	20	0	0	15	25	55	377	39	95	91	92	534	0.63
4	20	5	6	11	17	22	59	9	14	35	36	116	0.44
0	0	7	15	8	8	5	60	11	21	28	29	108	0.63
1	1	33	372	2	3	20	91	13	25	50	65	492	0.73
3	7	3	7	6	14	25	94	13	19	46	48	153	0.52
1	2	6	9	10	11	57	239	45	155	101	113	418	0.61
3	7	3	8	4	11	6	14	11	12	24	24	56	0.44
6	11	3	3	11	12	45	95	72	90	125	282	213	0.30
19	39	68	203	61	88	248	1,276	170	387	472	1,410	2,020	0.59
21	89	1	2	0	0	4	7	15	19	38	60	117	0.45
1	1	1	1	1	1	1	1	16	22	23	23	48	0.38
								19	27	47	85	165	0.49
								8	8	36	36	168	0.65
								55	173	108	116	432	0.59
								11	21	31	32	104	0.43
								0	0	18	47	51	0.47
								38	57	89	90	223	0.53
								13	108	70	115	301	0.60
								151	249	320	334	947	0.50

Figure 2: Usage of Overloading (Magic Methods).

System	Files		Magic Methods							GC
	MM	WI	S	G	I	U	C	SC		
CakePHP	18	18	5	12	7	0	10	0	0.28	
CodeIgniter	4	5	1	5	0	0	1	0	0.32	
DoctrineORM	4	4	1	1	Table 9: Usage of Magic Methods					
Drupal	2	13	0	1	System				Files	
Gallery	26	26	4	15					Total	VA
Joomla	10	10	2	7	CakePHP				640	213
Kohana	2	2	2	2	CodeIgniter				147	24
MediaWiki	14	14	2	3	DoctrineORM				501	112
Moodle	61	1,030	27	41	Drupal				268	99
osCommerce	0	0	0	0	Gallery				505	166
PEAR	1	1	0	0	Joomla				1,481	999
phpBB	0	0	0	0	Kohana				432	67
phpMyAdmin	2	2	1	1	MediaWiki				1,480	656
SilverStripe	9	9	3	5	Moodle				5,367	2,002
Smarty	7	8	5	6	osCommerce				529	84
SquirrelMail	0	0	0	0	PEAR				74	48
Symfony	6	6	2	1	phpBB				269	155
					phpMyAdmin				341	148
					SilverStripe				514	328
					Smarty				126	26

Table 6: Derivability of Variable-Variable Names.

System	Variable-Variable Uses		
	Total Names	Derivable	Derivable %
CakePHP	20	19	95.0
CodeIgniter	20	16	80.0
Drupal	1	1	100.0
Gallery	7	2	28.6
Joomla	2	0	0.0
Kohana	7	5	71.4
MediaWiki	11	5	45.5
Moodle	39	29	74.4
osCommerce	89	0	0.0
PEAR	1	1	100.0
phpBB	82	62	75.6
phpMyAdmin	112	86	76.8
SilverStripe	3	1	33.3
Smarty	40	38	95.0
SquirrelMail	24	10	41.7
WordPress	37	28	75.7

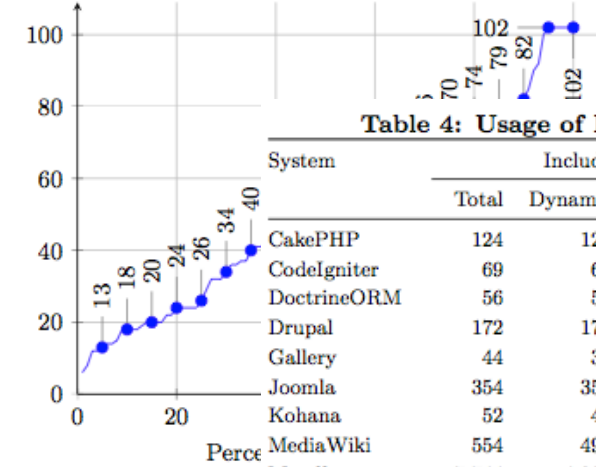


Figure 3: Features Needed to Implement a Feature. The feature count is on the x-axis and Implemented Features is on the y-axis.

Table 4: Usage of Dynamic Includes.

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

BitAnd, BitOr, BitXor, BoolAnd, BoolOr, Concat, Div, Equal, Geq, Gt, Identical, LShift, Leq, LogAnd, LogOr, LogXor, Lt, Minus, Mod, Mul, NotEqual, NotId, Plus, RShift, toArray, toBool, toFloat, toInt, toObject, toString, toUnset, break, continue, declare, do, exit, expStmt, for, foreach, goto, haltCompiler, if, label, return, suppress, switch, ternary, throw, tryCatch, while, classConstDef, classDef, closure, const, functionDef, global, include, interfaceDef, methodDef, namespace, propertyDef, static, traitDef, use, call, eval, methodCall, shellExec, staticCall, fetchClassConst, fetchConst, fetchStaticProperty, propertyFetch, traitUse, var, empty, instanceof, isSet

Table 3: Usage of eval and create_function.

System	Files			Total Uses	Gini
	Total	EV	WI		
CakePHP	640	3	3	5/1	0.33
CodeIgniter	147	2	2	3/0	0.17
DoctrineORM	501	0	0	0/0	N/A
Drupal	268	1	1	1/0	N/A
Gallery	505	5	7	1/4	0.00
Joomla	1,481	6	7	7/1	0.21
Kohana	432	3	3	1/2	0.00
MediaWiki	1,480	5	5	4/1	0.00
Moodle	5,367	39	1,077	34/30	0.30

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

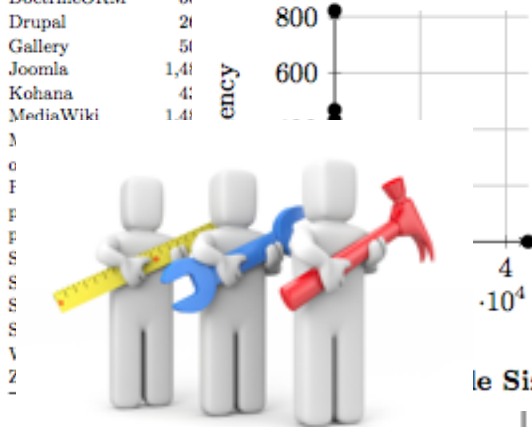
Table 1: The PHP Corpus.

System	Version	PHP	Release Date	File Count	SLOC	Description
CakePHP	2.2.0-0	5.2.8	2012-07-02	640	137,900	Application Framework
CodeIgniter	2.1.2	5.1.6	2012-06-29	147	24,386	Application Framework
Doctrine ORM	2.2.2	5.3.0	2012-04-13	501	40,870	Object-R
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 M
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	Applicati
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 L
osCommerce	2.3.1	4.0.0	2010-11-15	529	44,952	Online R
PEAR	1.9.4	4.4.0	2011-07-07	74	31,257	Compon
phpBB	3	4.3.3	2012-01-12	269	148,276	Bulletin
phpMyAdmin	3.5.0	5.2.0	2012-04-07	341	116,630	Database
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	Templat
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	Applicati
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	Applicati

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

Table 10: Usage of Invocation Functions.

System	Files	CUF	CUFA	CUM	CUMA	Gini
	Total	Inv	Inc			
CakePHP	6					
CodeIgniter	1					
DoctrineORM	5					
Drupal	2					
Gallery	5					
Joomla	1,4					
Kohana	4					
MediaWiki	1,4					



osCommerce	95.1%	96.4%	SilverSt
ZendFramework	93.2%	97.3%	phpMyAd

Table 6: Derivability of Variable-Variable Names.

System	Varia	Total Names
CakePHP	20	
CodeIgniter	20	
Drupal	1	
Gallery	7	
Joomla	2	
Kohana	7	
MediaWiki	11	
Moodle	39	
osCommerce	89	
PEAR	1	
phpBB	82	
phpMyAdmin	112	
SilverStripe	3	
Smarty	40	
SquirrelMail	24	
WordPress	37	



What's New in PHP 5.5



By: Patrick Mulvey | Posted: July 12, 2013 | News & Opinion



Figure 5: Usage of Variable Features.

Table 9: Usage of Variadic Functions.

System	Files	VDefs	VCalls	LCalls	Gini
CakePHP	640	213	227	36	2,543
CodeIgniter	147	24	26	6	106
DoctrineORM	501	112	112	35	316
Drupal	268	99	108	23	503
Gallery	505	166	170	24	722
Joomla	1,481	999	1,048	15	8,537
Kohana	432	67	67	17	178
MediaWiki	1,480	656	688	90	5,036
Moodle	5,367	2,002	2,410	86	11,168
osCommerce	529	84	106	0	201
PEAR	74	48	48	1	643
phpBB	269	155	165	6	1,291
phpMyAdmin	341	148	148	5	1,135
SilverStripe	514	328	334	39	994
Smarty	126	26	29	0	109

Figure 3: age. The

</

Table 3: Usage of ev

System	Files	EV	Total	EV
CakePHP	640	0	640	0
CodeIgniter	147	0	147	0
Drupal	501	0	501	0
Gallery	268	1	268	1
Joomla	505	5	505	5
Kohana	1,481	6	1,481	6
MediaWiki	432	3	432	3
Moodle	1,480	5	1,480	5
osCommerce	5,367	39	5,367	39
PEAR	1,077	1,077	1,077	1,077
phpBB	34/30	0.30		

```
if ($data['type'] == "dropdown") {
    $count = $data['number'];
    echo "<select name='".$data['name']."'>";
    foreach ($call as $c) {
        echo "<option value='".$data['default']."'>";
        echo ">";
    }
}
```

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

Table 1: The PHP Corpus.

System	Version	PHP	Release Date	File Count	SLOC	Description
CakePHP	2.2.0-0	5.2.8	2012-07-02	640	137,900	Application Framework
CodeIgniter	2.1.2	5.1.6	2012-06-29	147	24,386	Application Framework
Doctrine ORM	2.2.2	5.3.0	2012-04-13	501	40,870	Object-R
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 M
Joomla	2.5.4	5.2.3	2012-05-02	1,481	152,218	CMS
Kohana	3.2	5.3.0	2011-07-25	432	27,230	Applicati
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 L
osCommerce	2.3.1	4.0.0	2010-11-15	529	44,952	Online R
PEAR	1.9.4	4.4.0	2011-07-07	74	31,257	Compon
phpBB	3	4.3.3	2012-01-12	269	148,276	Bulletin
phpMyAdmin	3.5.0	5.2.0	2012-04-07	341	116,630	Database
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	Templat
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	Applicati
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	Applicati

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

Table 10: Usage of Invocation Functions.

System	Files			CUF	CUFA	CUM	CUMA	Gini
	Total	Inv	Inc					
CakePHP	6							
CodeIgniter	1							
DoctrineORM	5							
Drupal	2							
Gallery	5							
Joomla	1,4							
Kohana	4							
MediaWiki	1,4							
Moodle	5,3							
osCommerce	5							
PEAR								
phpBB	2							
phpMyAdmin	3							
SilverStripe	5							
Smarty	1							
SquirrelMail	2							
Symfony	2,1							
WordPress	3							
ZendFramework	4,3							

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

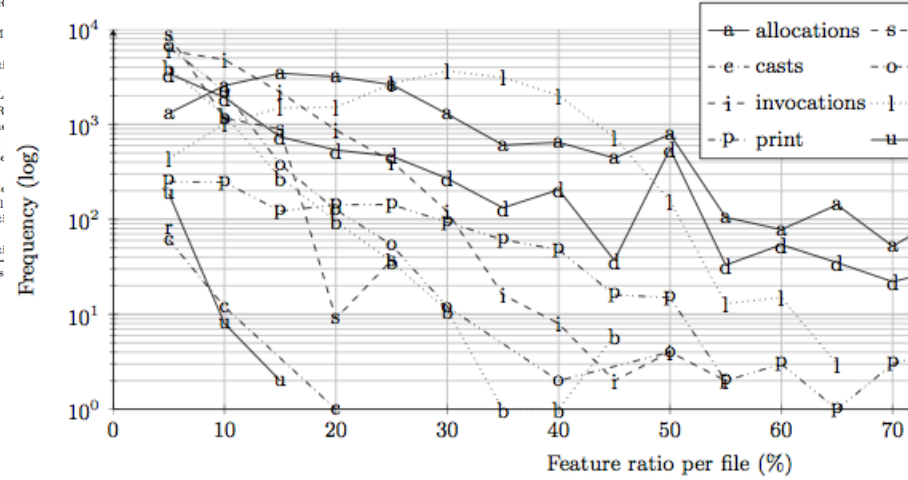
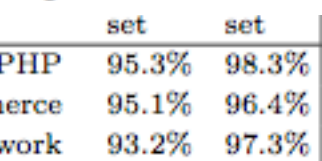


Table 5: Usage of Variable Features.

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
7	20	0	0	15	25	55	377	39	95	91	92	534	0.63
4	20	5	6	11	17	22	59	9	14	35	36	116	0.44
0	0	7	15	8	8	5	60	11	21	28	29	108	0.63
1	1	33	372	2	3	20	91	13	25	50	65	492	0.73
3	7	3	7	6	14	25	94	13	19	46	48	153	0.52
1	2	6	9	10	11	57	239	45	155	101	113	418	0.61
3	7	3	8	4	11	6	14	11	12	24	24	56	0.44
6	11	3	3	11	12	45	95	72	90	125	282	213	0.30
19	39	68	203	61	88	248	1,276	170	387	472	1,410	2,020	0.59
21	89	1	2	0	0	4	7	15	19	38	60	117	0.45
1	1	1	1	1	1	1	1	16	22	23	23	48	0.38
								19	27	47	85	165	0.49
								8	8	36	36	168	0.65
								55	173	108	116	432	0.59
								11	21	31	32	104	0.43
								0	0	18	47	51	0.47
								38	57	89	90	223	0.53
								13	108	70	115	301	0.60
								151	249	320	334	947	0.50

Figure 2: Usage of Variable Features.

Table 6: Derivability of Variable-Variable Names.

System	Variable-Variable Uses		
	Total Names	Derivable	Derivable %
CakePHP	20	19	95.0
CodeIgniter	20	16	80.0
Drupal	1	1	100.0
Gallery	7	2	28.6
Joomla	2	0	0.0
Kohana	7	5	71.4
MediaWiki	11	5	45.5
Moodle	39	29	74.4
osCommerce	89	0	0.0
PEAR	1	1	100.0
phpBB	82	62	75.6
phpMyAdmin	112	86	76.8
SilverStripe	3	1	33.3
Smarty	40	38	95.0
SquirrelMail	24	10	41.7
WordPress	37	28	75.7

Table 7: Usage of Overloading (Magic Methods).

System	Files		Magic Methods						GC
	MM	WI	S	G	I	U	C	SC	
CakePHP	18	18	5	12	7	0	10	0	0.28
CodeIgniter	4	5	1	5	0	0	1	0	0.32
DoctrineORM	4	4	1	1					
Drupal	2	13	0	1					
Gallery	26	26	4	15					
Joomla	10	10	2	7					
Kohana	2	2	2	2					
MediaWiki	14	14	2	3					
Moodle	61	1,030	27	41					
osCommerce	0	0	0	0					
PEAR	1	1	0	0					
phpBB	0	0	0	0					
phpMyAdmin	2	2	1	1					
SilverStripe	9	9	3	5					
Smarty	7	8	5	6					
SquirrelMail	0	0	0	0					
Symfony	6	6	2	1					
WordPress	4	4	0	0					

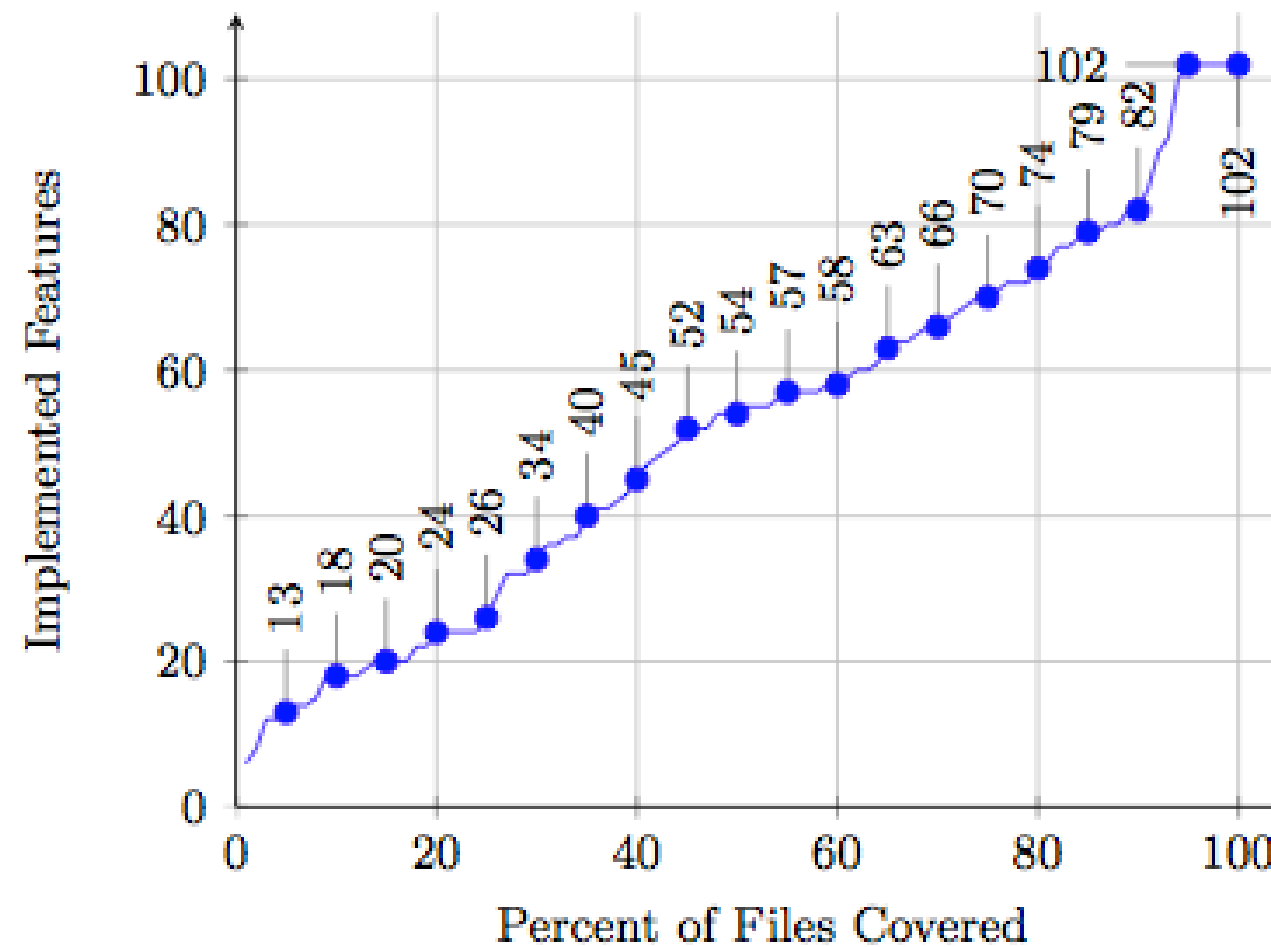
Table 9: Usage of Variadic Functions.

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

Table 3: Usage of eval and create_function.

System	Files			Total Uses	Gini
	Total	EV	WI		
CakePHP	640	3	3	5/1	0.33
CodeIgniter	147	2	2	3/0	0.17
DoctrineORM	501	0	0	0/0	N/A
Drupal	268	1	1	1/0	N/A
Gallery	505	5	7	1/4	0.00
Joomla	1,481	6	7	7/1	0.21
Kohana	432	3	3	1/2	0.00
MediaWiki	1,480	5	5	4/1	0.00
Moodle	5,367	39	1,077	34/30	0.30

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 dynamism. Everything can be modified, from the fields and methods of an object to its parents. This presents a challenge to static analysis.

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

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 *PRuby*, an extension to Diamondback Ruby (DRuby), a static type inference system for Ruby. *PRuby* augments DRuby with a novel dynamic analysis and transformation that allows us to precisely type uses of highly dynamic constructs. *PRuby*'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

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 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>