East Carolina UNIVERSITY

Variable Feature Usage Patterns in PHP

Mark Hills

30th IEEE/ACM International Conference on Automated Software Engineering
November 9-13, 2015
Lincoln, Nebraska, USA

http://www.rascal-mpl.org

Background & Motivation

An Empirical Study of PHP Feature Usage (ISSTA 2013)

An Empirical Study of PHP Feature Usage

A Static Analysis Perspective

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

ABSTRACT

PHP is one of the most popular languages for server-side application development. The language is highly dynamic, providing programmers with a large amount of flexibility. However, these dynamic features also have a cost, making it

languages, as of January 2013 ranking 6th on the TIOBE programming community index [5], used by 78.8 percent of all websites whose server-side language can be determined [4], and ranking as the 6th most popular language on GitHub [3]. PHP is dynamically typed, with a single-inheritance class

- Research questions:
  - How do people actually use PHP?
  - What assumptions can we make about code and still have precise static analysis algorithms in practice?

One focus area: variable features

- Core idea: identifier given as expression, computed at runtime
- One common use: prevent code duplication
- Also, allows identifier names to be part of configuration for plugins and extensions

```
if (is_array(${$x})) {
 ${$x} = implode($join[$x], array_filter(${$x}));
}
```

Where can variable features appear?

- Variables
- Function calls
- Method calls
- Object instantiations
- Property lookups
- Class constants
- Static method calls (target class, method name)
- Static property lookups (target class, property name)

How often do they occur in real programs?

- Not an uncommon feature
- So, cannot just make imprecise assumptions; at least one use in many files, although uses tend to be clustered (hence the Gini scores)
- Makes many analyses less precise: write through a variable feature could write to many different named entities (variables, properties, etc), call of variable feature could call many named functions or methods

All		
Files	Uses	Gini
92	490	0.60
42	157	0.46
25	131	0.70
50	502	0.73
47	155	0.52
241	940	0.57
30	69	0.42
428	1,043	0.46
143	255	0.31
631	2,501	0.54
52	175	0.49
23	48	0.38
51	148	0.48
37	112	0.52
120	607	0.68
32	108	0.44
18	51	0.47
94	268	0.56
77	246	0.43
213	548	0.49

Not being replaced by newer features (SANER 2015)

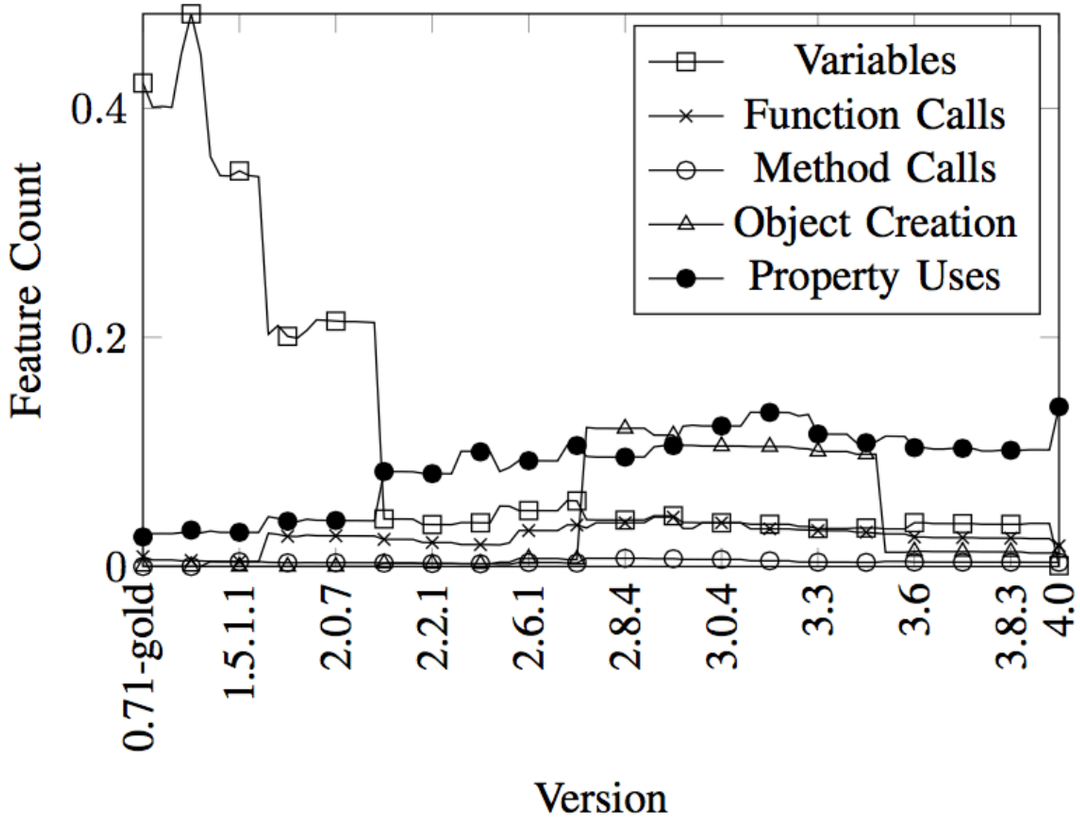


Fig. 1. Variable Features in WordPress, Scaled by SLOC.

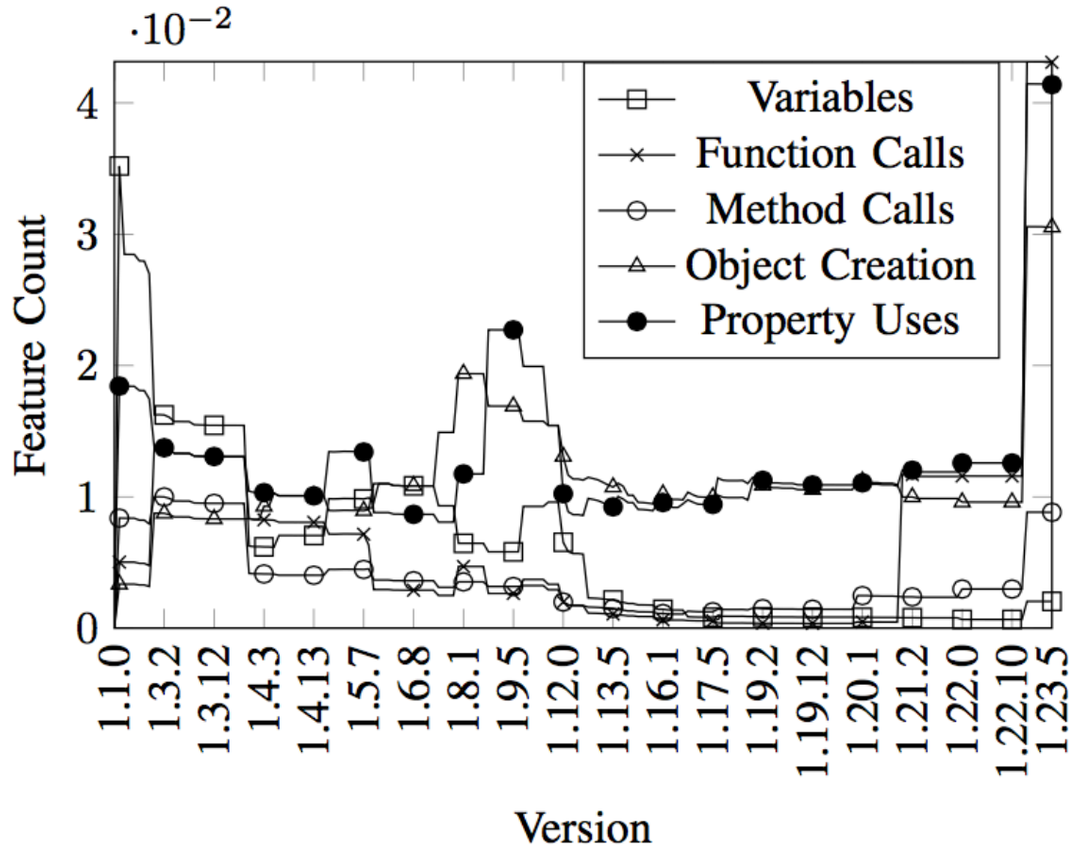


Fig. 2. Variable Features in MediaWiki, Scaled by SLOC.

- Some variable features are becoming less common (variable variables), some are going up (variable properties)

- No overall trend towards declining use, very system dependent

One insight: they often occur in patterns

```
$fields = array( 'views', 'edits', 'pages', ‘articles',
                 'users', 'images' );
foreach ( $fields as $field ) {
  if ( isset( $deltas[$field] ) && $deltas[$field] ) {
    $update->$field = $deltas[$field];
  }
}


foreach (array('columns', 'indexes') as $x) {
 if (is_array(${$x})) {
  ${$x} = implode($join[$x], array_filter(${$x}));
 }
}
```

One insight: they often occur in patterns

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
ZendFramework	7	5	71.4

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

- Mentioned in ISSTA'13
- But, only investigated manually, based on examining variable occurrences in the corpus, though that this *could* be automated

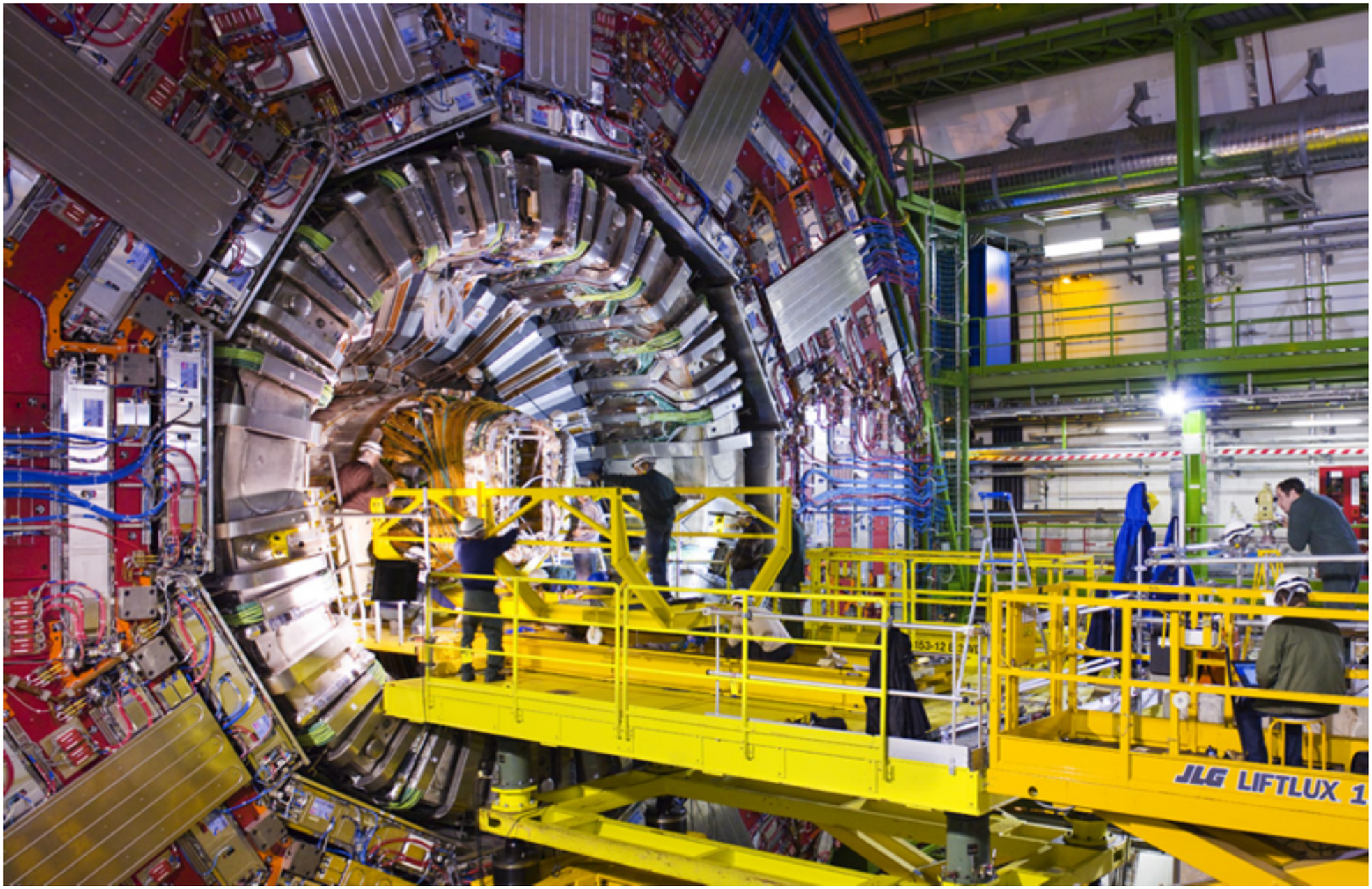
Research questions

- Do recognizable patterns of variable feature usage actually occur in real systems?

- If so, can we devise a lightweight analysis, guided by these patterns, to resolve occurrences of variable features in PHP scripts?

- Can we estimate how many occurrences of these features cannot be resolved statically?

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, Magento, MediaWiki, Moodle, Symfony, Zend

- Multiple domains: app frameworks, CMS, blogging, wikis, eCommerce, webmail, and others

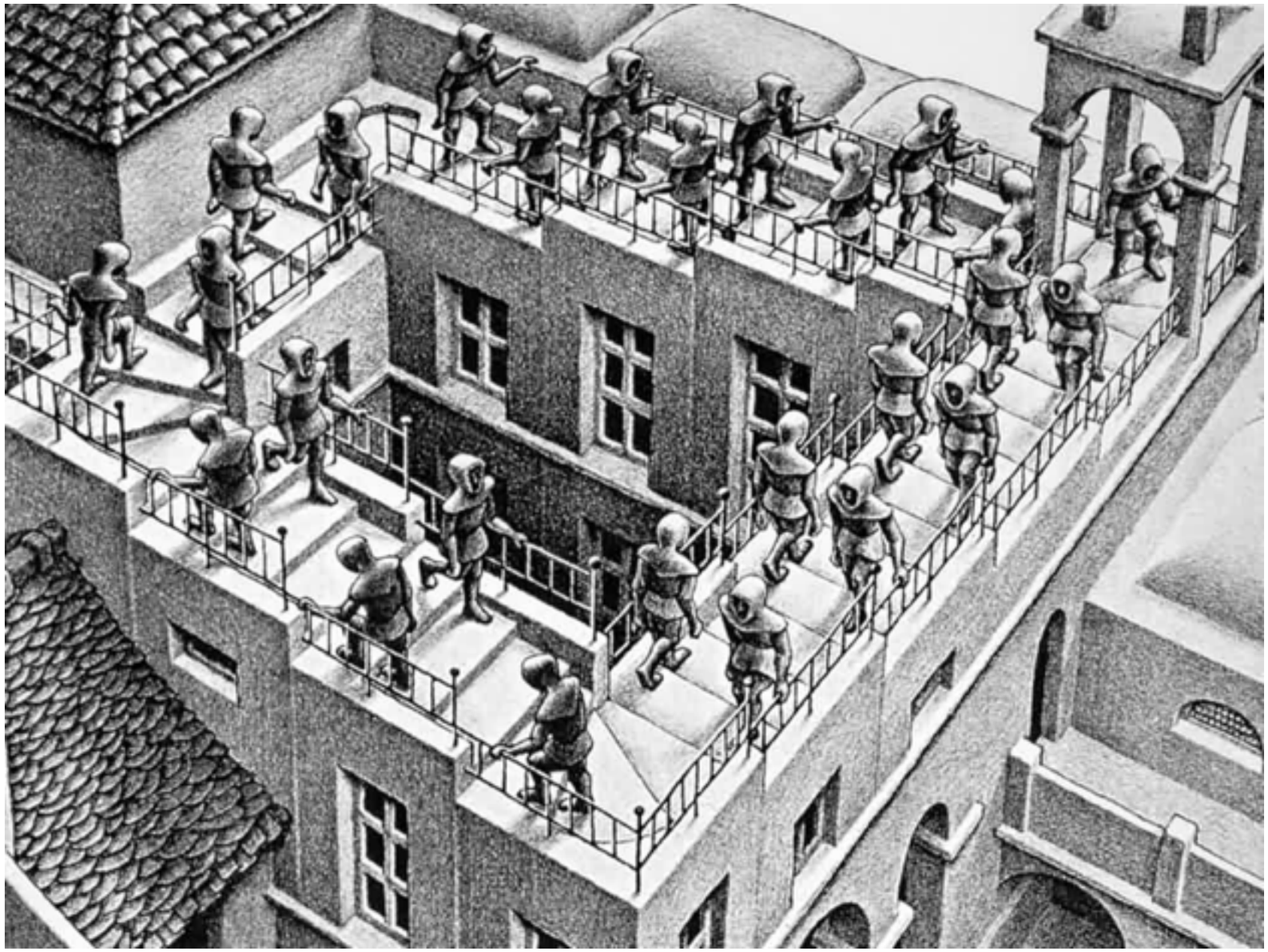
- Selected based on Ohloh rankings, based on popularity and desire for domain diversity

- 20 open-source PHP systems, 3.73 million lines of PHP code, 31,624 files

Methodology

- Corpus parsed with an open-source PHP parser
- Variable features identified using pattern matching
- Pattern identification and analysis scripted individually for each pattern using PHP AiR framework
- Patterns “ordered” (with more specific tried first), we don’t attempt to resolve already-resolved occurrences
- All computation scripted, resulting figures and tables generated
  - http://www.rascal-mpl.org/

Defining and Resolving Usage Patterns



Variable Feature Usage Patterns

- Focus on common patterns of usage for variable features
    - Loop patterns: identifier computed based on foreach key/value or for index (14 patterns total)
    - Assignment patterns: identifier computed based on local assignments into variable (4 patterns total)
    - Flow patterns: identifier provided by, or resolvable by, non-looping control flow comparisons (5 patterns total)
- Not all uses follow a pattern we have defined

Loop patterns: a first example

```
// MediaWiki, /includes/Sanitizer.php, lines 424-428
$vars = array( 'htmlpairsStatic', 'htmlsingle',
  'htmlsingleonly', 'htmlnest',
  'tabletags', 'htmllist', 'listtags',
  'htmlsingleallowed', 'htmlelementsStatic' );
foreach ( $vars as $var ) {
  $$var = array_flip( $$var );
}
```

Loop Pattern 2: Foreach iterates over array of string literals assigned to array variable, value variable used directly to provide identifier

Loop patterns: a second example

```
// WordPress, /wp-includes/ID3/getid3.php, lines 345-358
foreach (array('id3v2'=>'id3v2', ...)
    as $tag_name => $tag_key) {
    ...
    $tag_class = 'getid3_'.$tag_name;
    $tag = new $tag_class($this);
    ...
}
```

Loop Pattern 7: Foreach iterates directly over array of string literals, intermediate uses key variable to compute new string, intermediate then used to provide identifier

Loop patterns: a third example

```
// SquirrelMail,/src/options_highlight.php,lines 339-341
for ($i=0; $i < 14; $i++) {
    ${"selected".$i} = '';
}
```

Loop Pattern 13: For iterates over numeric range, string literal and loop index variable used as part of expression directly in occurrence to compute identifier

Assignment patterns: an example

```
// WordPress,/wp-includes/class-wp-customize-setting.php,
// lines 334-361 (parts elided for space, see paper)
switch( $this->type ) {
  case 'theme_mod' :
    $function = 'get_theme_mod';
    break;
 default :
    ...
    return ...
}
// Handle non-array value
if ( empty( $this->id_data[ 'keys' ] ) )
 return $function($this->id_data['base'],$this->default);
```

Assignment Pattern 1: String literals assigned into variable, variable used directly to provide identifier

Flow patterns: an example

```
// WordPress, /wp-includes/capabilities.php,
// lines 1054-1332
switch ( $cap ) {
    ...
    case 'delete_post':
    case 'delete_page':
        ...
        $caps[] = $post_type->cap->$cap;
        ...
    }
    ...
}
```

Flow Pattern 3: Switch/case switches on variable with literal cases, variable used directly to find identifier

How did we come up with these patterns?

- Look at uses in real code in the corpus to get ideas
- Extrapolate based on existing patterns (e.g., “we’ve seen this pattern with the foreach value, maybe it occurs with the foreach key as well”)
- Refine and/or discard based on attempts to use

Are these patterns effective?

- Loop patterns: 2485 of 8554 occurrences, 422 resolved, variable variables often resolved, can resolve some variable properties

- Assignment patterns: 5386 of 8554 occurrences, 396 resolved, patterns may be over-broad; resolution does better with method and function calls, but many unresolved

- Flow patterns: 2945 of 8554, 218 resolved; resolution quite good in limited cases (variable variables and properties in some systems)

- Overall: 13.3% resolved, including 40.8% of variable variables and 29.5% of variable methods, loop patterns most helpful

- Many occurrences match patterns, but resolution rate is fairly low

Can we improve these results?

- Some uses are truly dynamic, how can we tell if that is the case?
- Key idea: maybe usage patterns can help here too — are there patterns that indicate that a use is truly dynamic?

Anti-patterns

- Note: not programming anti-patterns, don’t indicate bad feature use

- Instead, indicate cases where we probably cannot resolve, feature is supposed to be dynamic
  - Identifier computation based on input parameter
  - Identifier computation based on function or method result (note: this may include functions we can simulate…)
  - Identifier computation based on one or more global variables

Measuring anti-patterns

- Anti-patterns computed similarly to patterns, but no ordering is given

- For each, two types of measurements
  - How many variable feature occurrences match an anti-pattern?
  - How many of these could we resolve anyway?

- Good anti-patterns should have a low number for the second, if we can resolve it then the anti-pattern has very low predictive power

Anti-pattern results

- Anti-patterns seem to have good predictive power
  - Roughly 9% of matches are resolved, 91% not resolved
  - 8554 variable feature occurrences total, 1137 resolved, 7717 unresolved
  - Anti-patterns find 5889 of these (roughly 72%)
- Room for improvement, but a good start, indicates that many unresolved occurrences probably cannot be resolved

Threats to validity

- Results could be very system specific (mitigation: varied corpus)
- There may be additional patterns that we have not discovered (but at some point, may be so uncommon we don't want to include it)
- A stronger analysis could resolve more variable features (but would lose useful information about the patterns)

Research questions, revisited

- Do recognizable patterns of variable feature usage actually occur in real systems? **YES**, many uses fall into the defined patterns

- If so, can we devise a lightweight analysis, guided by these patterns, to resolve occurrences of variable features in PHP scripts? **YES**, at least for the patterns we have investigated here, although resolution success is dependent on both the pattern and the feature type

- Can we estimate how many occurrences of these features cannot be resolved statically? **YES**, we believe anti-patterns help us to identify cases that cannot be resolved statically (are truly dynamic), even with a stronger analysis

Summary

- We've presented a number of patterns of usage for variable features in PHP and seen that many occurrences actually fall into these patterns

- We've seen that, in some cases, we can exploit these patterns to statically determine more precise sets of actual identifiers

- We have strong indications that many unresolved occurrences may actually be dynamic

Discussion

Thank you!
Any Questions?

- Rascal: http://www.rascal-mpl.org
- Me: http://www.cs.ecu.edu/hillsma

East Carolina University