Does It Scale? Symbolic Execution of PHP Web Applications

ABSTRACT

Dynamic web applications have become widely popular and are to a large proportion based on the scripting language PHP. Output approximation of web applications enables a range of additional tool support as well as possibilities for vulnerability detection. Unfortunately, recent approximation approaches have only been evaluated for smaller systems. This paper presents an experience report about the scalability of output approximation using symbolic execution of state-of-the-art PHP web applications. For a symbolic execution engine extended with support for object-oriented programming and arrays, we identified language features and corresponding programming patterns that impede symbolic execution and limit the scalability of this approach. Our findings include: (1) Dynamic features such as functions and includes are prone to fail for certain programming patterns. (2) Expressions containing elements from I/O, databases or files can heavily impede symbolic execution. Our findings provide useful guidelines to design new tools and also to improve the development process of statically analyzable web applications.

CCS CONCEPTS

•**Computer systems organization Embedded systems;** *Redundancy;* Robotics; •**Networks** Network reliability;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:

. 2016. Does It Scale? Symbolic Execution of PHP Web Applications. In *Proceedings of ACM Woodstock conference, Santa Barbara, California, USA, July 2017 (ISSTA'17),* 9 pages.

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the emerging world wide web, dynamic web applications have become widely popular. Various different implementation techniques have emerged, ranging from technologies for programming languages (JSP, ASP .NET) and web application frameworks for script languages (Ruby On Rails, Django) to languages tailored specifically to the domain of web applications. PHP [2] is a programming language focused on server-side application development. As of 2012, it was used by 78.8 percent of the ten million most popular websites (according to Alexa popularity ranking) [4], ranking 7th on the TIOBE programming community index [5], and was the ranked as the 6th most popular language on GitHub [3].

ISSTA'17, Santa Barbara, California, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

One common property of all technologies for dynamic web applications is *staged computation*: A dynamic web application as a whole consists of both static code, such as scripts, and dynamically generated code, such as client page output. The latter code, although assembled at runtime, may contain client-side parts of the web application, such as JavaScript. So, to study dynamic web applications in its entirety, we need to consider both static as well as dynamic aspects of systems.

Symbolic execution is a static analysis technique between program testing and program proving. For a program, symbolic values are used instead of concrete inputs [9, 14]. The underlying concept is to map program input to program output: Symbolic values are used and propagated throughout the symbolic execution and keep dependencies for program output traceable. This analysis technique helps unfold the *staged* nature of dynamic web applications. Since every feasible path can be executed the symbolic output contains both invariant output as well as output variants that depend on program input.

Knowledge about different output variants is leveraged by a number of analyses for the domain of web applications. Previous work presented tools for detecting and locating HTML validation errors [19], computing program slices across server-side and client-page code [17], or easing development and maintenance by extending IDE support for web applications with code navigation [16, 18]. Moreover, knowledge of output variants can increases code coverage for output-oriented testing [6].

Despite the various use cases for analyses based on an approximated output model, static output approximation for PHP web applications using symbolic execution has so far only been evaluated for small systems that are not maintained anymore. The tools presented by previous work [16–19] though are only practical, if for a given system the symbolic execution engine is scalable, i.e., will also approximate output accurately for larger and more recent systems with acceptable time and space consumption.

To investigate the question, whether we can have a practical and scalable symbolic executoin engine, we re-implemented the engine with the specifications of the previous symbolic execution semantics [16] for PHP and additional features, such as object-oriented programming. We evaluate our symbolic execution engine for several large-scale and modern PHP systems.

During the introduction of new semantics, we addressed two tradeoffs between accuracy of our execution and performance: For method calls with ambiguous targets as well as concrete execution of loops the number of program states can become infeasible. This may require additional engineering adjustments to tame the state space explosion at cost of accuracy.

Based on observations we identify conceptual limitations of symbolic execution for PHP. Dynamic features such as indirect function calls and include expression require concrete information to be evaluated properly. Since these expressions may be assembled at runtime and can contain symbolic information, for many cases this

restricts a symbolic execution engine from further execution. Although some effort to statically approximate include expression can be spent, it is often non-trivial since expressions can contain information coming from various inputs including databases, user inputs or configuration files. We empirically evaluate our observations for a corpus including large, actively maintained PHP systems.
Our key contribution in this paper include (1) a new tool infrastructure to statically approximate the client-page output for PHP web applications using symbolic execution and (2) a report of observations for state-of-the-art web applications as well as an empirical evaluation of conceptual limitations of symbolic executions for PHP web applications.

2 STATE OF THE ART

This section recaps the idea of symbolic execution, the tools which make use of it, and both conceptual and language-related limitations.

2.1 Symbolic Execution

Symbolic execution is a static analysis technique that was proposed by King [14] in 1976. Symbolic execution allows to explore feasible paths in a program. Based on normal program execution, the execution semantics is extended to handle symbolic values. An execution becomes symbolic by assuming symbolic values as program inputs rather than obtaining concrete values [9, 14]. These symbolic values are then propagated and used throughout the execution.
Execution starts with a tautology path condition. The control-flow can be split whenever the decision between branches is ambiguous. This is the case, when conditional expressions, for example for if-statements, evaluate to a symbolic value rather than an actual result. The corresponding path conditions (guards for instance) are conjoined with the previous path condition [14]. Additionally, the conjoined path condition can be checked for satisfiability in order to exclude infeasible paths.

2.2 Static Output Approximation for PHP

Symbolic execution is not dedicated to a particular programming language. For dynamic web applications, in our case for web applications written in PHP, it can be used to approximate all possible client page output variants. Before we present in detail existing tools that aim to ease development and maintenance of PHP web applications, we illustrate how output approximation using symbolic execution is pursued by starting with a small example.
Figure 1 shows two code listings. The snippet of server-side web application code in Figure 1a contains both HTML code (input form in lines 1 – 4) as well as PHP code: First, the function `make_titles` defined in lines 7 – 13 prints the string `$arg` passed as an argument between zero and an arbitrary number of times. Second, the function is indirectly called using the built-in mechnism `call_user_func` in line 15 with the string `Headline` and `4` as arguments. Finally, the input from the form in lines 1 – 4 is used: The variable `$greeting` consists of three concatenated strings, `Hello`, the value of input field `name`, and an exclamation mark. For the greeting there are two possible client page output variants: If the input field `name` is empty at runtime, line 19 is executed and `No name entered!`

```
1  <form method="POST">
2    <input type="text" name="name">
3    <input type="submit" value="Submit">
4  </form>
5
6  <?php
7  function make_titles($arg, $n) {
8    $is = shuffle(array(1,2,3));
9    while ($n < $is[0]) {
10     echo "<h1>" . $arg . "</h1><br />";
11     $n = $n + 1;
12   }
13 }
14
15 call_user_func("make_titles", "Headline", 4);
16
17 $greeting = "Hello " . $_POST['name'] .  "!";
18 if (!isset($_POST['name'])) {
19   echo "No name entered!";
20 } else {
21   echo $greeting;
22 }
23 ?>
```

(a) Snippet of server-side HTML and PHP web application code.

```
1  <form method="POST">
2    <input type="text" name="name">
3    <input type="submit" value="Submit">
4  </form>
5
6  // #repeat n < shuffle(is)
7  <h1>Headline</h1><br />
8  // #endrepeat
9
10 // #if !isset(name)
11 No name entered!
12 // #else
13 Entered α!
14 // #endif
```

(b) Approximated client page output of the server-side code in Figure 1a. The directives in lines 6 and 8 highlight output that can be repeated an arbitrary number of times, the directives in line 10 and 14 mark two alternative output parts.

Figure 1: Illustration of static ouptut approximation using symbolic execution

is printed, or, if a name was entered, line 21 is executed and the greeting constructed in line 17 is printed.
The following symbolic execution semantics for PHP features based on previous work [16] illustrate the necessary modifications and extensions made to handle the level of ambiguity that comes with handling symbolic values. In addition to normal output, the output approximation contains preprocessor directives to express repeatable parts as well as output variants. All approximated client page output corresponding to Figure 1a consisting of two different variants is illustrated in Figure 1b.
First, for loops in the previous execution semantics [16] only one iteration is executed instead of an arbitrary number of iterations. Since a loop condition can contain symbolic values, it might not be feasible to determine whether, and if so, when a loop execution terminates. Hence, any output constructed during the single loop iteration is highlighted (`repeat ...endrepeat`) to possibly be

repeated an arbitrary number of times. The output in line 7 is marked to be repeated an arbitrary number of times, as the number of loop iterations cannot be determined statically.
Second, a similar procedure applies to recursive functions. Once called, a recursive function is also only executed to recursion depth one per call, and any subsequent invocation of that function returns a symbolic value. This, again, is due to the difficulty to determine the exact recursion depth similar to the number of loop iterations.
Third, any form of user interaction providing input data as well as any form of input referred to from the deployment context of the web application, i.e. configuration files or databases, is assumed to be symbolic. The context variable `$_POST` represents input submitted to the server and, hence, is symbolic as every program input is per definition substituted by symbolic values. Finally, the output in lines 11 and 13 respectively are marked as alternative variants (`if ...else ...endif`) as symbolic execution of lines 18 – 22 in Figure 1a will explore both different branches.

2.3 Symbolic-Execution-based Tools For PHP

All tools proposed so far leverage a representation of all possible HTML client page output. Since any single client page output variant can be analyzed this enables tool support addressing the web application as a whole in spite of its staged nature, where parts of the web application are generated dynamically. Based on the approximated output representation, subsequent analyses can be conducted:

- *PhpSync*: Using all variants of HTML client page output, every single one can be statically checked for markup validity, i.e., if the web page conforms to syntactical specifications for HTML and other client-side languages. After the tool detects validation errors, auto-fixes can be provided. Otherwise, it traces validation errors back to source code responsible for the defect, which then can be refactored manually [19].
- *WebSlice*: Program slices enable to extract and understand the impact of changes in an application. To consider all of a dynamic web application for program slices, client page output needs to be taken into account. WebSlice combines PHP data-flow information with an output approximation to enable program slices across different languages [17].
- *Varis*: Editor services such as "jump to declaration" are nontrivial for dynamic web applications due to their staged nature. Varis provides editor services across stages for client-side code: Starting with an ouput approximation a callgraph with conditional edges is constructed, which allows navigation in client-site code (HTML, JS and CSS) although it is embedded in server-side code [16, 18].

2.4 Assumptions And Limitations

In addition to the assumptions introduced by symbolic execution, such as symbolic inputs and one-time loop execution, the symbolic execution engine described in previous work [16] does not support advanced language features, including object-oriented programming. Moreover, the range of standard library functions supported are mostly string operations, yet PHP provides a large collection of functions for array operations. All three analyses described in Section 2.3 though have only been applied to systems that are small and not maintained any more. Despite the functional benefits these tools provide to developers, their practicality for large and modern systems with respect to advanced language support has not been evaluated.

3 SCALABLE OUTPUT APPROXIMATION?

The main objective of this experience report is to investigate whether we can build a scalable symbolic execution engine as it is the underlying technique for static output approximation in previous work [16–19]. Our vision is a practical symbolic execution, i.e., it computes an accurate approximation for large and modern PHP systems with reasonable time and space consumption. We approach this goal by re-implementing the symbolic execution semantics of PhpSync [16] and extending it with support for additional language including object-oriented programming and array operations. We aim to support symbolic execution for a real-world example system of reasonable size that incorporates the missing language features described in Section 2.4. WordPress is a popular open-source Content Management System (CMS) with around 300.000 lines of code that provides a vivid plug-in environment and is used. The full implementation of Oak, the test suite as well as or test corpus (see Section 4.1) can be found at http://www.github.com/smba/oak.
For our implementation of a symbolic execution engine for PHP, Oak, we have chosen a test-driven approach with continuous integration using regression tests based on WordPress and SchoolMate (see Section 4.1). For SchoolMate, a rather straightforward and simple system, which we chose to compare our approximation results to those of PhpSync, we could achieve similar output approximation results. For WordPress though, we did not achieve high code coverage or an accurate approximation for WordPress as we faced a number of obstacles hindering symbolic execution in different ways: The following Subsection 3.1 documents our design choices we made, apparent trade-offs we recognized, and limitations we faced on our way towards a scalable symbolic execution engine; and what we have learned from bypassing limitations faced in spite of manual effort for future work. Subsection 3.2 compares our observations and results to *PhpSync* and presents the motivation for our empirical evaluation of our tool symbolic execution engine in Section 4.

3.1 Experience Report

3.1.1 Symbolic/Concrete Loop Execution. Throughout the implementation we basically followed the symbolic execution semantics of PhpSync [16]. As described in Section 2.2 for loops, only one iteration is executed since the number of iterations is indeterminable for most cases. While this is the case for while and do-while loops, the number of loop iterations for for or foreach loops is generally determinable and known before entering the loop body. For loops use a loop counter that usually is incremented or decremented after each loop iteration until a loop condition is not satisfied anymore; foreach loops simply iterate over a given array value. Our execution semantics follows this scheme and simulates concrete execution for those types of loops if possible. Otherwise, for instance, if a loop counter, loop counter limit for for loops, or variable for foreach loops is symbolic, we resort to executing only a single iteration.

This design choice was made since it is more precise and, in contrast to only considering one iteration in all cases, slightly increased code coverage for WordPress.

3.1.2 Method Target Space Explosion. For method invocations that for concrete execution have only one target object, exploring multiple execution paths can cause the space of target objects to grow rapidly. A minimal example is shown in the listing in Figure 2a where two different objects can be assigned to variable `$person` in line 12 and 14. For the method invocation in line 16 hence, the method is executed twice in total since we have two alternative target objects. To address this problem, we implement program state merging for symbolic execution. For the example in Figure 2a, after the control flow is split in line 11 into two branches, we merge both branches together subsequently. For both branches two different values have been assigned to variable `$person` under different path conditions respectively. We implement state merging inspired by work of Sen et al. [22] using value summaries. A value summary can be conceived as a value type. For state merging in our example, in the value summary, each value (or each object in our case) is associated to the path condition under that it has been assigned to a variable. This would allow to only pass the value summary to the method and only invocate it once, but for our execution to be sound, all objects would need to be of the same class type. If for instance, in our example there would be another class definition with a method named `greet` and an instance of that class is assigned to variable `$person` in one of the two branches, the method invocation in line 16 actually needs to be executed for two different target objects of a different type.
Therefore, for our execution engine to be sound, we were bound to execute invocated methods for multiple targets since we cannot assert that all values contained in summary are of the same type. Nevertheless, to scale the target object space, in addition, we implemented execution modes only allowing execution of a method for a subset of possible targets and assuming symbolic return values for the remaining targets. This execution mode was necessary for WordPress for several, but not all entry points that we studied, in order to terminate. All subsequent analyses in this paper do not use this execution mode, yet it is provided in out implementation.

3.1.3 Dynamic Language Features. Aside from the previous issues, we identified two dynamic language features that we were not able to execute accurately. Dynamic features in PHP are features that take as input rather a expression that is resolved at runtime than a static constant value. For WordPress, we faced missing files and functions that are included using dynamic includes and dynamic function invocation respectively.
First, file inclusion in PHP usually is managed by statements like `include`, `include_once` or `require` and `require_once`. For transitive includes, statements ending with `_once` will not include the same file twice. The include statements make an attempt to include a file, but do not return a warning in contrast to the require statements. Each of these statements takes as input either a string constant (static include) or an expression that is evaluated at runtime (dynamic include). The string that the expression evaluates to represents an absolute or relative paths to the target file that the interpreter finally attempts to include. Figure 2b shows a small example of a dynamic include containing information retrieved from a database. Since any input for a symbolically executed application is symbolic this information can be propagated to include expressions. Also, configuration files represent only one particular configuration or even just default values or placeholders. Hence, symbolic execuion is prone to fail for dynamic includes if symbolic information is contained.
Second, function can either be called directly by statically providing the name of the function (or method), or indirectly. Indirect function calls in PHP are enabled by using built-in functions like `call_user_func`, `call_user_func_array` or simply passing a string value with a direct function call to the `eval` function. The first two functions take as arguments the name of the function two call, and all arguments to that functions either as additional arguments or a single associative array respectively. The `eval` function evaluates a given PHP expression passed as a string literal. Again, we encountered issues similar to dynamic include expressions with indirect function call mechanisms: The example in Figure 2c illustrates two functions `header_serif` and `header_sserif` which both print a HTML headline. Depending on whether the value of the variable `$style` retrieved from a database is `serif` or `sserif`, a different function is called. Given that the function name is concrete at runtime, the desired function can be called. Otherwise, if the function name is assembled at runtime and contains symbolic values, no function can be called as the name is ambiguous or unknown. For this example, symbolic execution is not able to determine which function to call indirectly.

3.1.4 Symbolic/Concrete Input. In the previous subsection, we argued that dynamic includes and dynamic function invocations hinder symbolic execution due to imprecise resolution of expression if symbolic information is contained. For callable code parts as included scripts or invocated functions, we attempted to bypass this limitation by manually providing proper concrete input.
Consider the code example in Figure 2c where a function name could not be resolved precisely due to a missing string literal that is contained from a database query result. By manually providing either option `serif` or `sserif`, or both as alternatives, we are able to increase code coverage and the accuracy of our approximation with manual effort by making dynamic statements static. Of course we did not provide proper input exhaustively for every occurrence of dynamic features, yet this illustrates a basis for possible future work towards scalable symbolic execution. This manual approach addresses the same problem as dynamic test input generation for web applications, where proper concrete input is assumed rather than symbolic values to increase code coverage. We present work on this problem in Section 6.

3.2 Experience Results

The symbolic execution engine PhpSync [16] so far does not face the problems we documented in Section 3.1. The evaluation of PhpSync, i.e., the PHP systems that the tool was applied to, did neither contain object-oriented features, nor dynamic features like dynamic includes or dynamic function invocation with symbolic values.
As the observations so far did only refer to WordPress as a large-scale system, we wanted to study the impeding impact of dynamic features to symbolic execution for a wider selection of PHP systems

```php
<?php
class Person {
   function __construct__($name) {
      $this->name = $name;
   }
   function greet($greeting) {
      echo $greeting . $this->name;
   }
}

if (...) {
   $person = new Person("Alice");
} else {
   $person = new Person("Bob");
}
$person->greet("Good Morning, ");
```

(a) Dynamic dispatch: Multiple targets for method calls

```php
<?php
define("ROOT", getcwd());
define("TEMPLATES", ROOT . "templates/")

$template = mysql_result(...);

require_once TEMPLATE . $template;
```

(b) Dynamic Include Resolution

```php
<?php
function header_serif($title) {
   $style = "font-family: serif;";
   echo "<h1 style = '$style'>$title</h1>";
}
function header_sserif($title) {
   $style = "font-family: sans-serif;";
   echo "<h1 style = '$style'>$title</h1>";
}

$font_style = mysql_result(...);
call_user_func('header_' . $font_style, 'Title');
```

(c) Function calls by indirect invocation mechanisms

Figure 2: Code Examples of defective code features

systematically apart from the systems that PhpSync was applied to for two reasons.
First, the usage of dynamic features, not only of includes and function invocations, varies from system to system [10].
Second, we were not able to obtain results for all entry points possible for WordPress and hence could not compare our approximation results to those of PhpSync where client page output approximation is evaluated for all entry points. Therefore, we decided to systematically evaluate our symbolic execution engine for systems apart from WordPress, which is documented in the following Section 4.

4 EVALUATION

As already stated in the previous Section 3.1, we encountered several problems during the implementation of our symbolic execution engine for WordPress. The main items of critique of the symbolic execution engine and tools described in previous work [16–19] are the missing support for widely-used language features, and the evaluation only using small-scale systems. Although evaluating a tool for a small systems can be a valuable proof-of-concept, it does not necessarily cover the question of whether a tool is practical on a larger-scale. Therefore in the following we present our methods to evaluate practicality of our symbolic output approximation. Moreover, we investigate to which extent the conceptual limitations encountered with WordPress apply for a wider choice of PHP systems.

4.1 Experiment Setup

We have encountered several issues for WordPress that conceptually limit the scalability of symbolic execution. Since our observations so far only considers one example system an, we evaluate our symbolic execution engine for a wider range of PHP systems with regard to the following questions. The metrics to approach those questions are explained in Sections 4.3 and 4.4.
First, the evaluation of the output approximation using the symbolic execution engine, PhpSync, in previous work [16] has shown high code coverage for small systems. As our symbolic execution re-implements PhpSync and extends with support for additional language features, we ask: Can we replicate high coverage results at least on a small-scale. Therefore, our PHP corpus contains four small-scale systems used in previous work [16].
Second, we ask :Do system size and usage of modern language features factors affect the practicality of our symbolic execution engine? We selected both a range of large-scale and small-scale modern PHP systems to better separate and understand the circumstances under which conceptual limitations might affect practicality. For the selection of large-scale systems we borrow three systems from a case study addressing the feature usage in PHP systems. Some more detailed description of this case study can be found in Section 6. For the small-scale systems we selected five systems from a list of recent Content Management Systems (CMS) written in PHP [1]. For both selections we were bound to select a smaller number of system than we intended to since the parser used in our implementation, Quercus[1], did not support a number of language features. The full list of PHP systems for that we evaluate our symbolic execution engine along with descriptive code metrics can be found in Table 1.

4.2 Measuring Approximation Success

In order to accurately answer the question how good our symbolic execution engine approximates client page output, we would require knowledge of all possible variants to compare our approximation against. Since we do not have any ground truth information alike, we approach this demand by using a heuristic. Rather than having knowledge about all client page output variants we say our approximation is accurate if all string literals (which are embedded in and scattered across server-side code) that may eventually become part of any client page output variant (in the following referred to as *output candidate*) is contained in our symbolic output approximation. It is non-trivial to determine whether a string literal is an output candidate, so we heuristically classify string literals as output candidates if they contain the characters <, >, or both since we expect output to contain HTML tags.

[1] Quercus footnote

System	Version	Classification	SLOC	#files	#OCs	#includes
AddressBook	8.2.5.2		51,907	239	1009	186
SchoolMate	1.5.4		8,118	65	853	88
TimeClock	1.04		20,800	63	7920	306
WebChess	1.0.0		5,219	28	470	56
Drupal	7.5.0	CMS	52,464	125	3569	749
phpBB	3.1.9		327,371	1,398	3606	206
phpMyAdmin	4.6.3		303,582	871	7103	571
Anchor	0.12.1		15,054	201	987	32
Kirby	XXX	CMS	XXX	XXX	654	23
Automad	XXX	CMS	XXX	XXX	655	8
Monstra	XXX	CMS	XXX	XXX	1934	48
Nibbleblog	XXX	CMS	XXX	XXX	1013	28

Table 1: Corpus of twelve PHP systems. The file count includes files with a .php, .inc, .bit or .module extension.

We evaluated this heuristic manually with a sample of 400 string literals randomly selected from our corpus of PHP systems (see Table 1). We measured for our heuristic classifier a precision of 94 percent, and a recall of 50 percent. This means that six percent of the string literals are classified incorrectly as false positives. In turn, the classifier is highly distinctive as 96 percent of the string literals are classified correctly. The recall of 50 percent means that half of the string literals, which we manually classified as output candidates, actually were responsive to the classifier. We attempted to increase recall by looking for further distinctive properties to build a classifier from, but could not do so without decreasing precision. In spite of missing half of the output candidates, we decided to use a simple, yet precise and distinctive classifier. We rather work on a smaller sample subset of actual output candidates with less false positives than a larger set of string literals, where more are mistakenly classified as output candidates.

4.3 Measuring Approximation Accuracy

In Section 4.2 we introduced the definition of output candidates as expected output. For an approximation to be most accurate, it needs to contain all output candidates of a system analyzed. Based on whether output candidates are reached by an execution path and part of the output approximatiopn define two metrics to approximate measuring accuracy.

We measure how much of the expected output was actually processed by the symbolic interpreter. Once a line of code, a statement or expression containing an output candidate is actually an element in an execution path, we define this output candidate as *reached*. We define the metric *reach coverage* as the ratio of output candidates that are reached and the total number of output candidates in the analyzed system. Although a reached output candidate is processed by the symbolic interpreter, it does not guarantee we will see that output candidate in the symbolic output. Therefore, we define the metric *output coverage* as the ratio of output candidates that are contained in the output of the symbolic interpreter and the total number of output candidates in the analyzed system.

The coverage results for our case study are illustrated in Figure 3a. We could replicate high coverage results for the four small-scale systems with a reach coverage and output coverage over 80 percent respectively. For the three modern/large-scale systems we measured poor coverage with reach coverage ranging from 5 to 30 percent and output coverage ranging from 4 to 20 percent. We measured medium to high coverage for the more recent small-scale systems: reach coverage ranging from 43 to 85 percent, output coverage ranging from 40 to 85 percent.

4.4 Understanding Limitations

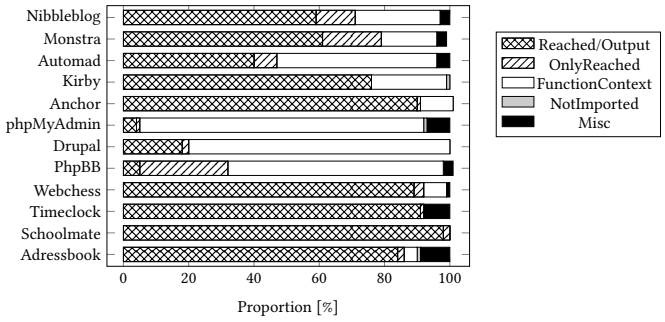
As we have seen in Section 4.3 both measured coverage metrics were poor for large-scale and small-scale/recent systems. To understand what output candidates we missed and why, we further investigated our approximation results and conducted three more measurements.

4.4.1 What literals did we miss? Our initial approach to understand missed output candidates is to find out whether their surrounding program code was not accessed/accessible, and if so, why. We are interested in those output candidates that are located in HTML files (possibly with nested PHP), or located in a function definition. Plain HTML files do not require any further execution and represent output candidates that just need to be included properly to be part of the output. Aside, output candidates that are part of a function definition are only missed if the corresponding function is never called at runtime.

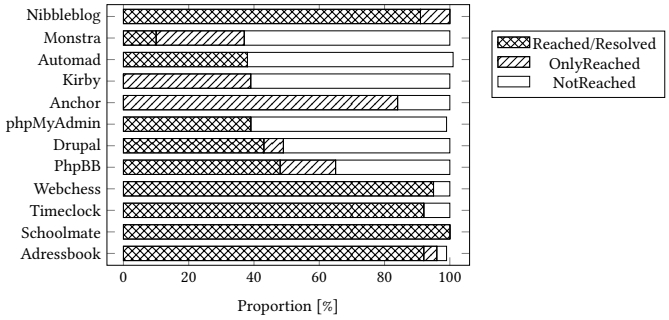
We started classifying missed output candidates by their string literal context. As the classification statistics in Figure 3a illustrate, for almost all systems tested (except for AddressBook and TimeClock) not-reached output candidates had a function context.

For both cases of context, either inclusion of a script file failed, or a function call failed. If an HTML file is not part of the output, it also is never reached, i.e., included. An include can fail due to various reasons, such as an imprecise evaluation of the include expression returning a symbolic value, or simply a missing file.

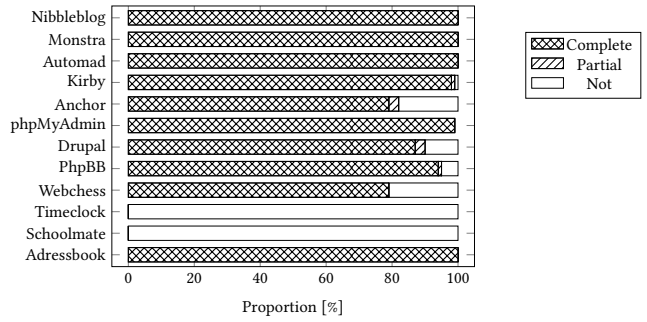
For a function to be never reached there are several scenarios: A function is undefined at runtime if the corresponding script file is never included. In turn, if the function is defined at runtime, the



(a)



(b) Include coverage and accuracy.



(c) Distribution dead function output candidates that are par- tially or completely explained due to callback candidates.

Figure 3: Results

function is dead code if there is no call site for this function, or the function is only called indirectly and the function name could not be resolved accurately. In PHP there are several ways to call a function beside direct call sites.

Given a symbolic value, indirect call mechanisms like callback-commands are likely to fail since target function name and the symbolic value do not match. This also applies for the evaluation of include expressions as concrete values representing include targets can be included, but any include target containing symbolic values is ambiguous. Since it is unlikely that all functions containing missed output candidates are dead code having no call site (direct or indirect) at all, in Section 4.4.3 we further investigate how many functions fail due to inaccurately resolved indirect function calls. Moreover, we evaluate the accuracy of evaluation of include expressions in the next section.

4.4.2 Why did includes fail? From Section 3.1.3 we learn that include expressions can not be evaluated accurately for symbolic values, and from the previous Section 4.4.1 that very little of missed output candidates can be explained by simply missing to include a file. To better understand the impact of include expression resolution throughout analyzing systems, we construct two additional metrics. First, we are interested in how many include expressions throughout the execution we actually reach in a system. Therefore we introduce the metric *reach coverage for include expressions* as the ratio of reached include expressions and the total number of reach expressions in a system to check whether failed includes may have caused subsequent include expressions and so on. Second, we measure how successful include expressions are resolved.

Figure 3b illustrates the classification of include expressions for each system: We can either reach an include expression and successfully resolve it, or not. Or, we did not reach an include expression at all. Figure 3b indicates that except for the small-scale systems as well as Anchor and Nibbleblog we did not really reach a great portion of include expressions, although for most systems the resolution success rate was over 80 percent. Note that one failed attempt to include a script file can result in a cascade of missing more include expressions as the include target script file is not included and executed.

4.4.3 Why did function calls fail? Aside from failed or successful includes, this metric describes one scenario of why functions containing not-reached output candidates are never called: The trivial explanation would be that there are no direct calls of those functions, yet PHP offers a number of ways to call a function indirectly, for instance using built-in functions, or simply evaluating strings as PHP code using the `eval` function.

Moreover we need to take into account that one missed function execution can result in missing even more function calls (direct or indirect) and executions. Thus, for functions containing not-reached output candidates we measure whether they can be (1) only, (2) partially or (3) never accessed transitively from functions that have no direct call sites. We refer to those functions (or access points) having no direct call site as callback candidates since the only way to reach them is through indirect mechanisms. Measuring how many not-reached output candidates can be explained by functions that are callback candidates helps to understand whether, and if so, indirect call mechanisms and their usage may impact our analysis' code coverage.

The callgraph snippets in Figure 4 illustrate the idea of detecting the functions' access points: The nodes at the top depict callback candidates, the nodes at the bottom with labels `X`, `Y`, and `Z` depict function definitions containing not-reached output candidates. In Figure 4 function `X` can only be accessed through callback candidates, as for Figure 4 function `Y` can be partially accessed by callback candidates, whereas function `Z` is never accessible through callback candidates.

For failed function calls to functions containing not-reached output candidates we measured how many of those functions were only, or partially accessibly through indirect calling mechanisms (see Section 3.1.3). As Figure 3c illustrates, around 80 percent (except

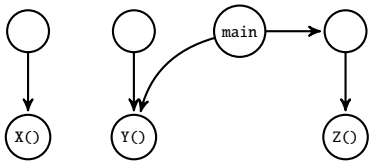


Figure 4: Callgraph analysis: The bottom nodes represent functions containing missed output candidates, the top nodes depict entry points to them.

for TimeClock) all missed output candidates of each system were only accessible through callback candidates, i.e., functions that are only accessible through indirect calling mechanisms.

5 LESSONS LEARNED

In this section we summarize the insights gained from our observations during the implementation as well as the from the results obtained by our experimental evaluation.

Dynamic features are problematic. From out observations with WordPress in Section 3 we learn that dynamic language features, in particular include expressions and indirect function calls, in combination with symbolic values cannot be evaluated accurately since concrete values are required. For our observations we were able to explain most of the missed output candidates with indirect function calls. For include expressions we are indeed able to resolve most expressions successfully, yet we do not distinguish between static and dynamic includes in this case study as this would exceed the scope of our paper. However, we believe it is plausible to assume that defective include expressions can cause a cascade of missing files and subsequent includes. Throughout the experiment we have seen that dynamic features, are prevalent for modern PHP systems as they serve a genuine purpose in the programming language. Nevertheless their usage in situations when combined with assumed symbolic values restricts symbolic execution and any tool leveraging results obtained from that.

Bypassing conceptual limitations. Symbolic execution is used to approximate client page output since it allows to unfold the staged nature of dynamic web applications. Aside from conceptual limitations in the systems studied, limitations can be addressed. To use approaches incorporating symbolic execution with dynamic web applications in the long run, more concrete dummy information, similar to test oracles, needs to be provided to decrease the number of assumed symbolic inputs and avoid dynamic features being confronted with symbolic values. Also, if the usage of tools using static output approximation with symbolic execution is intended, avoiding dynamic features that are prone to be inaccurately executed for symbolic execution may be an adjustment strategy to program in an analyzable way. Future work to increase code coverage and approximation quality might involve automated test case generation as tools like Apollo [7, 8] or previous work [28] to decrease the number of symbolic inputs and provide proper and concrete input variants.

Practical tools require practical evaluation. Last, we want to address the limited informative value of analyses and tools presented in previous work [15–19, 26] where only small PHP systems have been studied. These papers introduce techniques for static output approximation of PHP web applications [15, 16, 25] or propose tools based on respective approximations [16–19, 26, 27]. As these approaches address practical use cases, they require a more and further comprehensive evaluation in order to better understand possible limitations and assess their practicality, in particular for large-scale systems.

The road so far. For this paper, the extension of the symbolic execution PhpSync [16] support for object-oriented programming was not the main goal, yet necessary to symbolically execute modern PHP systems. The observations regarding object target space explosion for method calls documented in Section 3.1.3 are only anecdotal and, with the exception of WordPress, we did not notice any significant impact on scalability. Nevertheless, more research is required to better and systematically understand possible drawbacks of object-oriented symbolic execution. This question though is beyond the scope of this paper, as we aimed to explore limitations of the practicality and scalability of static output approximation using symbolic execution. Future work might address design guidelines to enable programming analyzable applications, tool support to scale the number of assumed symbolic inputs or systematically provide concrete data to increase the practical benefit of the tools described above.

6 RELATED WORK

Static Output Approximation. To approximate program slices for web applications, Ricca et al. [20, 21, 23] approximate dynamically generated output. For output generating statements, such as `echo` or `print`, all strings are unquoted (code extrusion). If those statements contain variables, these are linked to string concatenations using a proposed flow analysis called *string-cat propagation*. From these flows representing approximated output subsequent program slices are computed.
Minamide [15] approximates client page output of web applications by describing possible output by a context-free grammar that is constructed statically from the PHP code for a given regular expression of user input. The constructed grammar enables analyses such as detecting cross-site scripting vulnerabilities by checking whether user input has been sanitized, and HTML validation by determining whether the constructed grammar is contained in a depth-bound HTML grammar. Based on Minamide's approximation approach several vulnerability analyses addressing cross-site scripting [27] and SQL injection [26] have been proposed. Wang et al. [24] utilizes this string analysis to detect strings visible at the browser and enable internationalization of web applications.
Another approach is proposed by Wang et al. [25], where output for a web application is approximated using a hybrid approach: A dynamic webpage is executed with concrete input and the execution is recorded at runtime. Changes in the client-side output then can be mapped to corresponding PHP code using static impact analysis.

Dynamic Features. According to a case study by Hills et al. [12] of the feature usage in PHP, based on a corpus of state-of-the-art-projects, dynamic includes are less frequently used than static includes, but usage frequency is varying from system to system.

Further work by Hills et al. [11, 13] approaches static resolution of dynamic includes using both context-insensitive resolution on file level by simplifying PHP constants; and context-sensitive on program level for transitive includes. In spite of promising results, these approaches, similar to our results, face limitations for truly dynamic includes if resolution is not sound due to information not being available in the source code like database query results.

Input Generation for PHP. Apollo [7, 8], automated test input generation using concolic testing [28], test case generation using search based software engineering [6]

7 CONCLUSION

REFERENCES

[1] https://codegeekz.com/12-lightweight-cms-for-building-websites/. (????). Accessed: 2016-09-30.

[2] PHP. http://php.net/. (????).

[3] PHP Usage on GitHub. https://github.com/languages/php. (????).

[4] PHP Usage Statistics. http://w3techs.com/technologies/details/p1-php/all/all. (????). Accessed: 2016-1-12.

[5] TIOBE Index for PHP. . (????).

[6] Nadia Alshahwan and Mark Harman. 2011. Automated Web Application Testing Using Search Based Software Engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 3–12. DOI:http://dx.doi.org/10.1109/ASE.2011.6100082

[7] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. 2008. Finding bugs in dynamic web applications. ACM Press, 261. DOI:http://dx.doi.org/10.1145/1390630.1390662

[8] S Artzi, A Kiezun, J Dolby, F Tip, D Dig, A Paradkar, and M D Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Transactions on Software Engineering* 36, 4 (July 2010), 474–494. DOI:http://dx.doi.org/10.1109/TSE.2010.31

[9] J. A. Darringer and J. C. King. 1978. Applications of Symbolic Execution to Program Testing. *Computer* 11, 4 (April 1978), 51–60. DOI:http://dx.doi.org/10.1109/C-M.1978.218139

[10] Mark Hills. 2015. Variable Feature Usage Patterns in PHP (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 563–573.

[11] Mark Hills and Paul Klint. 2014. PHP AiR: Analyzing PHP Systems with Rascal. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 454–457.

[12] Mark Hills, Paul Klint, and Jurgen Vinju. 2013. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 325–335. DOI:http://dx.doi.org/10.1145/2483760.2483786

[13] Mark Hills, Paul Klint, and Jurgen J Vinju. 2014. Static, lightweight includes resolution for PHP. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 503–514.

[14] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. DOI:http://dx.doi.org/10.1145/360248.360252

[15] Yasuhiko Minamide. 2005. Static approximation of dynamically generated Web pages. ACM Press, 432. DOI:http://dx.doi.org/10.1145/1060745.1060809

[16] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Building Call Graphs for Embedded Client-side Code in Dynamic Web Applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 518–529. DOI: http://dx.doi.org/10.1145/2635868.2635928

[17] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2015. Cross-language Program Slicing for Dynamic Web Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 369–380. DOI:http://dx.doi.org/10.1145/2786805.2786872

[18] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2015. Varis: IDE Support for Embedded Client Code in PHP Web Applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 693–696. http://dl.acm.org/citation.cfm?id=2819009.2819140

[19] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2011. Auto-locating and Fix-propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 13–22. DOI:http://dx.doi.org/10.1109/ASE.2011.6100047

[20] Filippo Ricca and Paolo Tonella. 2001. Web Application Slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01) (ICSM '01)*. IEEE Computer Society, Washington, DC, USA, 148–. DOI:http://dx.doi.org/10.1109/ICSM.2001.972725

[21] Filippo Ricca and Paolo Tonella. 2002. Construction of the System Dependence Graph for Web Application Slicing. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '02)*. IEEE Computer Society, Washington, DC, USA, 123–. http://dl.acm.org/citation.cfm?id=827253.827727

[22] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 842–853. DOI:http://dx.doi.org/10.1145/2786805.2786830

[23] Paolo Tonella and Filippo Ricca. 2005. Web application slicing in presence of dynamic code generation. *Automated Software Engineering* 12, 2 (2005), 259–288. http://link.springer.com/article/10.1007/s10515-005-6208-8

[24] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei, and Jiasu Sun. 2010. Locating need-to-translate constant strings in web applications. ACM Press, 87. DOI: http://dx.doi.org/10.1145/1882291.1882306

[25] Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. 2012. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 16. http://dl.acm.org/citation.cfm?id=2393614

[26] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, Vol. 42. ACM, 32–41.

[27] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 171–180. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4814128

[28] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. 2008. Dynamic Test Input Generation for Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 249–260. DOI: http://dx.doi.org/10.1145/1390630.1390661