Does It Scale? Symbolic Execution of PHP Web Applications

Stefan Mühlbauer
Technische Universität Braunschweig

ABSTRACT

Dynamic web applications have become widely popular and are to a large portion implemented in the scripting language PHP. Dynamic languages like PHP though challenge static analysis, as they generate code at run-time. One approach to unfold their staged nature and to approximate client page output is symbolic execution. Knowledge about possible output variants enables a range of developer tools as well as analyses for vulnerability detection. However, previous tools that leverage symbolically approximated output have only been evaluated for smaller systems and scalability for larger systems has not been assessed.

In this paper, we criticize the limited expressiveness of the evaluation of practicality for respective tools. We ask whether symbolic execution for output approximation for PHP is scalable, and what limitations can be faced. By extending a symbolic execution engine based on previous work and evaluating it for some larger PHP systems we find dynamic features to be likely to hinder symbolic execution. We explore strategies to mitigate limitations inspired by work on dynamic symbolic execution and aim to push for more comprehensive evaluation for developer tools.

1 INTRODUCTION

With the emerging world wide web, dynamic web applications have become widely popular. Various different implementation techniques have emerged, ranging from web 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 [3] is a programming language focused on server-side application development. As of 2017, PHP was used by 82.4 percent of the ten million most popular websites (according to Alexa popularity ranking) [24], ranking 10th on the TIOBE programming community index [4], and, in 2015, was the ranked as the 4th most popular language on GitHub [2].

A common characteristic of all dynamic web applications is their *staged nature*: 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 generated at run-time, may contain client-side parts of the web application, such as JavaScript. Therefore, to study dynamic web applications in its entirety, we need to consider both static as well as dynamic aspects of those systems.

Symbolic execution is a static analysis technique between program testing and program proving. For a program, symbolic values are assumed instead of concrete inputs [10, 15]. 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 explored and 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 tools and analyses for the domain of web applications. Previous work presented tools for detecting and locating HTML validation errors [20], computing program slices across server-side and client-page code [18], or easing development and maintenance by extending IDE support for web applications with code navigation [17, 19]. Moreover, knowledge of output variants can increase code coverage for output-oriented testing [5].

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 actively maintained. The tools presented by previous work [17–20] though are only practical, if for a given system the symbolic execution engine is scalable, i.e., will also approximate client page 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 execution engine, we re-implement the symbolic engine with the specifications of the previous symbolic execution semantics for PhpSync [17] for PHP and extend it with 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 for object-oriented programming we face a trade-off between accuracy and scalability: Whilst exploring multiple paths, for method invocations the number of target objects can grow rapidly and become infeasible. Moreover, based on observations we identify limitations of symbolic execution for PHP. For an accurate execution dynamic features, such as dynamic function invocations and dynamic includes, require precisely resolved include targets and function names respectively. Since these expressions can contain symbolic values at run-time, for many cases this restricts a symbolic execution engine from further execution as callable code sections become inaccessible. Although some effort can be spent to statically resolve dynamic include expressions [14], or to simply provide static counterparts manually, resolution is often not trivial since expressions can contain symbolic values assumed instead of concrete program input, including databases, user inputs or configuration files. We empirically evaluate our symbolic execution engine for a corpus including large and 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 practical limitations of symbolic executions for PHP web applications.

2 STATE OF THE ART

This Section recaps the idea of symbolic execution, the tools using output approximation based on it and whose evaluation we criticize as well as current conceptual and language-related limitations.

2.1 Symbolic Execution

Symbolic execution is a static analysis technique that was proposed by King [15] 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 [10, 15]. These symbolic values are then propagated and used throughout the execution.

Symbolic execution commences 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 branch predicates are conjoined with the previous path condition [15]. 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 `$arg` passed as an argument between zero and an arbitrary number of times. Second, the function is dynamically invocated using the built-in mechanism `call_user_func` in line 15 with the string `Headline` and `4` as arguments. Finally, the input from the form in lines 1 – 4 is used: The variable `$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 run-time, line 19 is executed and `No name entered!` is printed. Otherwise, if a non-empty 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 [17] 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 illustrated in Figure 1b 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 [17] only one iteration is executed instead of an arbitrary number of iterations. Since a loop condition can contain symbolic values, it might not

```
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 output approximation using symbolic execution

be feasible to determine whether, and if so, when a loop execution terminates. Hence, any output assembled 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 invocated, a recursive function is also only executed to recursion depth one per call, and any subsequent invocation of that function returns a symbolic value. Again, this 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 concrete 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 PHP context variable `$_POST` represents input submitted to the server and, hence, is symbolic as every program

input is per definition substituted by assumed 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 and presented in the following leverage symbolically approximated 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 developer tools and analyses have been proposed:

- *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, if possible, auto-fixes are provided. Otherwise, it traces validation errors back to source code responsible for the defect, which then can be refactored manually [20].
- *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 [18].
- *Varis*: Editor services such as "jump to declaration" are not trivial for dynamic web applications due to their staged nature. Varis provides editor services across stages for client-side code: Starting with an output approximation a call-graph with conditional edges is constructed, which allows navigation in client-site code (HTML, JS and CSS) although it is embedded in server-side code [17, 19].

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 [17] does not yet 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 tools described in Section 2.3 have only been applied to systems that are small and not actively maintained. Despite the functional benefits that these tools can provide to developers, their practicality for large and modern systems with respect to advanced language support has not been assessed.

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 [17–20]. 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 [17] 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 actively maintained. 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 chose 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.

3.1 Experience Report

We document in this Section the design choices we made, tradeoffs 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.

3.1.1 Symbolic/Concrete Loop Execution. Throughout the implementation we followed the symbolic execution semantics of PhpSync [17]. 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-loops or for-each 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; for-each 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 for-each 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 respectively. 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 `$person` under different path conditions respectively. We implement state merging inspired by work of Sen et al. [23] using value summaries. A value summary can be conceived as a value type. For state merging in our example, in the value summary, each value (or each object in our case) is associated to the path condition under that it has been assigned to a variable. This would allow to only pass the value summary to the method and only invocate it once. For our execution though to be sound, all objects would need to be of the same class type. If for instance, in our example there would exist another class definition with a method named `greet` and an instance of that class is assigned to variable `$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 are bound to execute invocated methods for multiple targets since we cannot assert that all values contained in the value 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 our implementation.

3.1.3 Dynamic Language Features. Aside from the previous issues, we identified two dynamic language features that, in many cases, we were not able to execute accurately. Dynamic features in PHP are features that take as input rather a expression that is resolved at run-time 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 run-time (dynamic include). The string that the expression evaluates to represents an absolute or relative path 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 assumed to be 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 execution is prone to fail for dynamic includes if symbolic values are contained.
Second, a function can either be invocated directly by statically providing the name of the function (or method), or indirectly. Indirect (or dynamic) function invocations 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 static function invocation to the `eval` function. The first two functions take as arguments

```
1  <?php
2  class Person {
3     function __construct__($name) {
4        $this->name = $name;
5     }
6     function greet($greeting) {
7        echo $greeting . $this->name;
8     }
9  }
10
11 if (...) {
12    $person = new Person("Alice");
13 } else {
14    $person = new Person("Bob");
15 }
16 $person->greet("Good Morning, ");
```

(a) Dynamic Dispatch : Multiple target objects for method invocation `greet()`

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

$template = mysql_result(...);

require_once TEMPLATE . $template;
```

(b) Dynamic Include Resolution: The value of `$template` is retrieved from a database query result.

```
<?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) Dynamic Function Invocation using `call_user_func`

Figure 2: Code examples for dynamic features

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 those with dynamic include expressions. 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 run-time, the desired function can be called. Otherwise, if the function name is assembled at run-time 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 dynamically.

3.1.4 Symbolic/Concrete Input. In the previous subsection, we argued that dynamic includes and dynamic function invocations

can hinder symbolic execution due to imprecise resolution of expressions if symbolic values are contained. For callable code sections as included scripts or 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 obtained 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 [17] 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 in a significant number.

As the observations so far did only refer to WordPress as a large-scale system, we want to study the hindering impact of dynamic features to symbolic execution for a wider selection of PHP systems 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 [11].

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 [17–20] 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 limitations encountered with WordPress apply for a wider choice of larger PHP systems.

4.1 Experiment Setup

We have encountered several problems for WordPress that conceptually limit the scalability of symbolic execution. Since our observations so far only consider 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 [17] 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 of the small-scale systems that PhpSync has been evaluated for [17].

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 limitations might affect practicality. For the selection of large-scale systems we borrowed three systems from a case study by Hills et al. [11] addressing the feature usage patterns and frequencies in PHP systems. Some more detailed description of this case study can be found in Section 6. For our choice of modern 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 systems 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. All measurements in the following are cumulated for all possible entry points of a system, i.e., each script file. The symbolic execution operates with a timeout of ten minutes and terminates if this timeout is exceeded.

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 is contained in our symbolic output approximation. In the following we refer to those expected output string literals as *output candidates*.

With respect to the staged nature of web applications it is not trivial to statically determine whether a string literal is an output candidate. Consequently, we heuristically classify string literals as output candidates if they contain the characters <, >, or both since we expect output to contain HTML tags.

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

[1] Quercus is an open-source Java implementation of PHP 5 by Caucho Technology. We reused the parser front-end for our implementation. Quercus can be obtained from http://quercus.caucho.com.

System	Version	Classification	SLOC	#files	#Output candidates	#Include Expressions
AddressBook	8.2.5.2	Web-based address and phone book	51,907	239	1009	186
SchoolMate	1.5.4	School administration	8,118	65	853	88
TimeClock	1.0.4	Web-based time tracking	20,800	63	7920	306
WebChess	1.0.0	Chess server software	5,219	28	470	56
Drupal	7.5.0	Content Management System	52,464	125	3569	749
phpBB	3.1.9	Forum	327,371	1,398	3606	206
phpMyAdmin	4.6.3	Web-based MySQL administration	303,582	871	7103	571
Anchor	0.12.1	Content Management System	15,054	201	987	32
Kirby	2.4.1	Content Management System	51.778	435	654	23
Automad	0.10	Content Management System	13.978	64	655	8
Monstra	3.0.4	Content Management System	41.102	490	1934	48
Nibbleblog	4.0.5	Content Management System	20.376	328	1013	28

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

turn, the classifier is highly distinctive as 94 percent of the output candidates are classified as such. The recall of 50 percent means that half of the string literals, which we manually classified as output candidates, actually are responsive to the classifier. We attempted to increase recall by looking for further distinctive properties to build a classifier from, but were not able to 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 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 approximation we 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. For our first evaluation question in Section 4.1 we could replicate high coverage results for the four small-scale systems with a reach coverage and output coverage over 80 percent respectively. Whereas 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 investigate our approximation results and conduct three subsequent measurements.

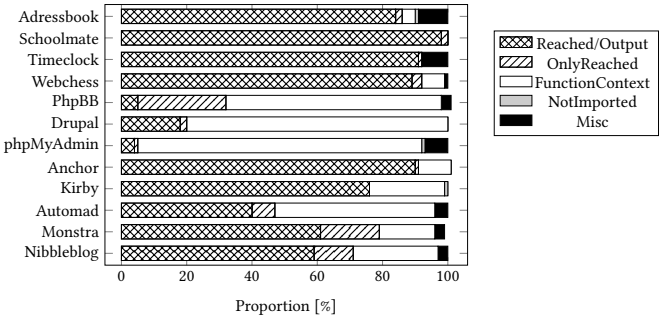
4.4.1 Where did we miss output? Our initial approach to understand missed output candidates is to find out whether their surrounding program code cannot not accessed, and if so, why. We are interested in those output candidates that are located in HTML files (possibly with nested PHP), or located in 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 invocated at run-time or simply just dead code.

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.

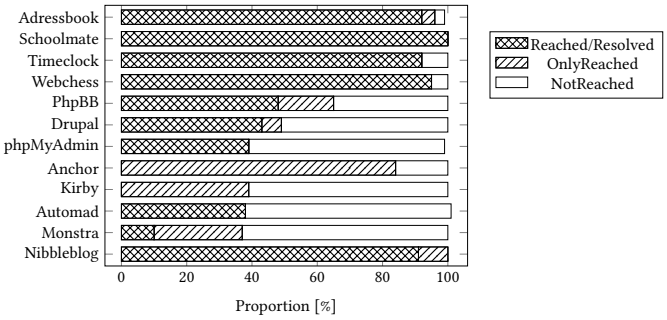
For both cases of context, either inclusion of a script file failed, or a function call failed. If an HTML file is not part of the output, it also is never reached, i.e., included. An include can fail due to various reasons, such as an imprecise evaluation of the include expression returning a symbolic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

For a function to never be reached there are several scenarios: A function is undefined at run-time if the corresponding script file is never included. In turn, if the function is defined at run-time, the function is dead code if there is no call site for this function, or the function is only called dynamically and the function name could not be resolved precisely. In PHP there are several ways to call a function beside static call sites, as described in Section 3.1.3.

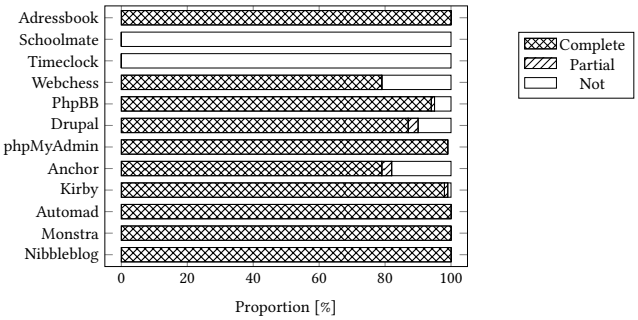
Given a symbolic value, indirect function invocation mechanisms like callback-commands are likely to fail since target function name



(a) Classification distribution of *reached* output candidates (Reached/Output, OnlyReached) and *not-reached* output candidates (FunctionContext, NotImported, Misc) during the analysis for each system.



(b) Classification distribution of *reached* output include expressions (Reached/Output, OnlyReached) and *not-reached* include expressions.



(c) Distribution of *not-reached* output candidates (in function contexts) that are only (Complete), partially (Partial), or not accessible via callback candidates.

Figure 3: Evaluation results for output candidates, include expressions and call-graph analysis for not-reached output candidates

and the symbolic value the expression evaluates to 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 (static or dynamic) at all, in Section 4.4.3 we further investigate how many functions fail due to inaccurately resolved indirect function invocations. Moreover, we evaluate the accuracy of evaluation of include expressions in the next section.

4.4.2 (Dynamic) Includes. From Section 3.1.3 we learn that include expressions containing symbolic values cannot be resolution precisely, 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 measure the number of reached include expressions in a system to check whether failed includes may have caused subsequent include expressions and so on.

Second, we measure how many reached include expressions are successfully resolved throughout the analysis for each system.

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 reach a great portion of include expressions, although for most systems the resolution success rate (ratio of resolved and reached include expressions) 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. This may explain the low number of reached include expressions, but relatively high success rate.

4.4.3 Dynamic Function Invocations. Aside from failed or successful includes, this metric describes one scenario of why functions containing not-reached output candidates are never invocated: The trivial explanation would be that there are no static invocations of those functions, yet PHP offers a number of ways to invocate a function dynamically, for instance using built-in functions, or simply evaluating strings as PHP code at run-time using the `eval` function.

Moreover we need to take into account that one missed function execution can result in missing even more function invocations. 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 having no static call site as *callback candidates* since the only way to reach them is through dynamic function invocation 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 function invocation mechanisms and their usage may impact our analysis' code coverage.

The call-graph 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

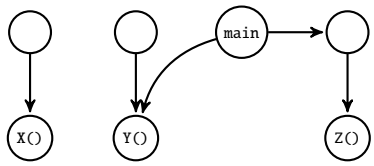


Figure 4: Call-graph analysis: The bottom nodes represent functions containing missed output candidates, the top nodes depict entry points to them.

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 invocations to functions containing not-reached output candidates we measured how many of those functions were only, or partially accessibly through indirect function invocation mechanisms (see Section 3.1.3). As Figure 3c illustrates, around 80 percent (except 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. Hence, we believe the impact of imprecisely resolved function names for dynamic function invocations, aside from dead code, has a limiting impact on symbolic execution since callable code sections become inaccessible.

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 dynamic function invocations containing symbolic values, often cannot be executed accurately since include targets or function names respectively are resolved imprecisely. For our observations we were able to explain most of the missed output candidates with indirect function invocations. For include expressions we are indeed able to resolve most reached 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 imprecisely resolved include expressions can cause a cascade of missing files and subsequent includes. Throughout the evaluation we have seen that dynamic features, are commonly used in modern PHP systems as they serve a genuine purpose in the programming language. Nevertheless their usage in situations where they contain 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 being limited by dynamic features, we exercised replacing dynamic features with respective static counterparts in order to bypass limitations by manually providing expressions that can be resolved precisely.To use approaches incorporating symbolic execution with dynamic web applications in the long run, more concrete input, 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. Existing work approaching this goal [6, 7, 30] incorporates dynamic symbolic execution, i.e. concrete execution alongside symbolic execution. 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.

Practical tools require practical evaluation. We wanted to address the limited informative value of analyses and tools presented in previous work [16–20, 28] where only small PHP systems have been studied. These papers introduce techniques for static output approximation of PHP web applications [16, 17, 27] or propose tools based on respective approximations [17–20, 28, 29]. 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 [17] 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. 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 practicality of the tools described above.

6 RELATED WORK

Symbolic Execution. Symbolic execution [10, 15] remained an conceptual approach for decades, but has been refined and extended over the last years. Several approaches incorporating constraint solving and leveraging concrete execution alongside symbolic execution emerged and enable automated test case generation; the scalability of these approaches is still challenged by path explosion and constraint solving [9].
Dynamic symbolic execution [9] incorporates symbolic execution alongside concrete execution and has shown promising results in automated test generation[6, 7, 30] for PHP web applications and might help mitigate the conceptual limitations faced with symbolic values and dynamic features for future work on output-oriented symbolic execution.
An overview about recent evolution of symbolic execution is provided by work of Cadar et al. [9], as well as a more tool-centered perspective [8].

Static Output Approximation. To approximate program slices for web applications, Ricca et al. [21, 22, 25] 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 [16] 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 [29] and SQL injection [28] have been proposed. Wang et al. [26] 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. [27], where output for a web application is approximated using a hybrid approach: A dynamic web page is executed with concrete input and the execution is recorded at run-time. 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. [13] 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, yet usage frequency varies from system to system. Further work by Hills et al. [12, 14] 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.

7 CONCLUSION

In this paper we have addressed symbolic execution of PHP web applications that is used for client page output approximation. These approximations are leveraged be several developer tools for PHP that yet have not been evaluated for larger systems. We re-implemented and extended an existing symbolic execution engine to analyze several real-world PHP applications with the intention to evaluate the scalability of symbolic execution for PHP. Our work illustrates the necessity of more comprehensive evaluation of developer tools as symbolic execution is a conceptually suitable, but not necessary practical approach to approximate client page output. Dynamic features of PHP in combination with symbolic values though can hinder precise symbolic execution and limit the scalability and practicality of tools relying on symbolically approximated output.

In the future we would like to apply dynamic symbolic execution for output approximation and possibly bypass existing limitations, as dynamic symbolic execution has shown promising results for automated testing and verification.

REFERENCES

[1] 2015. 12 Lightweight CMS for Building Websites. https://codegeekz.com/12-lightweight-cms-for-building-websites/. (June 2015). Accessed: 2017-01-15.

[2] 2015. PHP Popularity on GitHub. https://github.com/blog/2047-language-trends-on-github. (2015). Accessed: 2017-01-15.

[3] 2017. Official PHP Website. http://php.net/. (2017). Accessed: 2017-01-15.

[4] 2017. TIOBE Index for PHP. http://www.tiobe.com/tiobe-index/php/. (2017). Accessed: 2017-01-15.

[5] 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

[6] 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

[7] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Softw. Eng.* 36, 4 (July 2010), 474–494. DOI:http://dx.doi.org/10.1109/TSE.2010.31

[8] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 1066–1071. DOI:http://dx.doi.org/10.1145/1985793.1985995

[9] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. DOI:http://dx.doi.org/10.1145/2408776.2408795

[10] 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

[11] 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.

[12] 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.

[13] 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

[14] 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.

[15] 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

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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] W^{3}Techs Web Technology Surveys. 2017. Usage statistics and market share of PHP for websites. https://w3techs.com/technologies/details/pl-php/all/all. (2017). Accessed: 2017-01-15.

[25] 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

[26] 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

[27] 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

[28] 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.

[29] 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

[30] 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