Does It Scale? Static Output Approximation of PHP Web Applications Using Symbolic Execution

Stefan Mühlbauer
TU Braunschweig, Germany

Christian Kästner
Carnegie Mellon University, USA

Tien N. Nguyen
University of Texas, USA

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

Static program behavior, Static metrics, Dynamic language features, Static Analysis, PHP

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].

One common property of all technologies for dynamic web applications is *staged computation* of output: A dynamic web application as a whole consists of both static code, such as scripts, and dynamically generated code, such as responses to HTTP requests. 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, i.e., abstract, yet fixed inputs are applied instead of concrete inputs [6, 8]. The underlying concept is to map program input (symbolically) to program output. Different classes of inputs may result in variational output, representing dependencies of output and input. As symbolic execution of a program returns an input-output-mapping for symbolic inputs, this analysis technique helps unfold the *staged* nature of web applications: Since every feasible path is executed all possible output is contained in the symbolic output, and all dynamically generated variational parts of the web application can be analyzed.

Knowledge about different output variants enabled useful analyses for the domain of web applications, such as detecting and locating HTML validation errors [13]. As for developers, from an output model, outlines and callgraphs for easier IDE code navigation [10], or program slices [11] can be computed. Existing work has been integrated in the Varis plugin for the Eclipse IDE [12].

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 smaller systems that are not maintained anymore. The previously mentioned tools 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 good time and space consumption.

To investigate the question, whether we can build practical tools from the the symbolic execution engine, we re-implemented the engine with the specifications of the previous symbolic execution semantics [10] for PHP and additional features, such as object-oriented programming. We evaluated our symbolic execution engine for large-scale and modern PHP systems.

During introduction of new semantics for object-orientation we encountered two main challenges. First, we were bound to introduce more concrete execution features to the symbolic execution engine, e.g. foreach loops for arrays, as this increased code coverage significantly. Second, we had to make a trade-off between the accu-

ISSTA '17 July 9–13, 2013, Santa Barbara, CA, USA

DOI: 10.475/123_4

racy of method calls and the scalability of this language feature as a method call for different program state variants can have multiple targets.

Based on our observations and evaluation results, we learned that for dynamic PHP constructs, expressions that are assembled dynamically, in combination with symbolic values symbolic execution can be impractical. For object-orientation and method calls, symbolic execution semantics can be a tightrope walk between state space explosion and imprecision. Moreover, the nature of PHP challenges a practical symbolic execution engine since for it to be accurate, a large number of functions from the standard library needs to be supported.

Our key contributions in this paper include (1) a new tool infrastructure to statically approximate the output for PHP web applications using symbolic execution, and (2) a report of our observations with state-of-the-art web applications as well as an explanation on conceptual limitations for symbolic execution of PHP applications.

2. STATE OF THE ART

Symbolic execution is one of integral analysis techniques used by the tools whose practicality we aim to evaluate. This section recaps the idea of symbolic execution, the tools which make use of it, and both conceptual and language-related limitations of symbolic execution applied to PHP so far.

2.1 Symbolic Execution

Symbolic execution is a static analysis technique that was proposed by James C. King in 1976 [8]. Symbolic execution allows to explore all feasible paths in a program. Based on normal program execution, the execution semantics is extended to handle symbolic values, which are abstract, but fixed; an execution becomes symbolic by introducing symbolic values as program inputs instead of concrete values [6, 8]. These symbolic values are then propagated and used throughout the execution.

Execution starts with a plain path condition, usually a tautology. Whenever the control-flow can be split into different branches, the corresponding path conditions (guards for instance) are conjoined with the previous path condition [8]. Additionally, the conjoined path condition can be checked for satisfiability to exclude infeasible paths. Consequently, for a given program symbolic execution computes a mapping from (symbolic) input values to output values.

2.2 Workflow And Existing Tools

The work-flow for all existing tools is illustrated in Figure 2. Prior to any analysis, the web application is symbolically executed which for every execution path yields its corresponding client page output variant. Next, all client page output variants are parsed and merged into a compact representation of all client page output variants. This output model, also referred to as *VarDOM* (Variational Document Object Model), is similar to a DOM, yet it can contain symbolic value nodes and conditional node types to represent variants as alternatives.

All tools proposed so far leverage a representation of all possible HTML client page output. Since any variant can be analyzed in particular, this enables tool support addressing the web application as a whole in spite of its staged nature, where parts of the web application (client-side output) are generated dynamically.

Based on the approximated output representation subsequent analyses can be conducted, which have been implemented in the following tools:

- *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 validation errors are detected, auto-fixes are provided. Otherwise, the validation error can be traced back to source code responsible for the defect, which then can be refactored manually [13].
- *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 [11].
- *Varis*: Editor services such as "jump to declaration" are non-trivial 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 [10, 12].

```php
<?php
$x = $_POST['name'];
$y = $x + 1;
```

```php
<?php
function foo($arg) {
   echo $arg;
   foo($arg);
}
$z = foo(3);
```

```php
<?php
if ($z < 0) {
   $r = 'A';
} else {
   $r = 'B';
}
echo $r;
```

```php
<?php
while ($z > 0) {
   echo 'B';
   $z = $z - 1;
}
```

Figure 1: Example for symbolic execution of PHP code

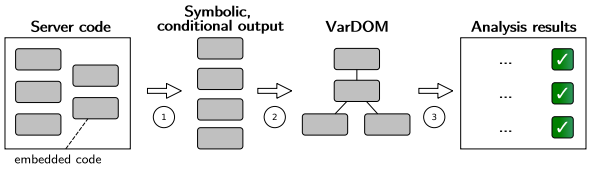

Figure 2: Bener

2.3 Assumptions and Limitations

- Design Decisions: Single loop and recursion execution due to conceptual limitations
- Conceptual Limitations: Symbolic input as form input, interaction
- Language-related or implementation-specific: No OOP, array support, standard lib

3. SCALABLE OUTPUT APPROXIMATION

4. EVALUATION

As already stated in the previous section, we encountered sevaral problems during the development of the symbolic interpreter. Therefore this section describes our methods to evaluate practicality of the output approximation, and possible explanations.

4.1 Measuring Approximation Success

For the evaluation of our output approximation we require a ground truth output model to compare our approximation against. Since output literals are embedded in server-side code and scattered, it is nontrivial to determine whether a string literal is part of the ouput or not. So, in the absence of ground truth for the approximated output we are bound to choose a *heuristic approach* in order to measure the accuracy of our aproximation. We define a string literal to be an *output candidate* if there exists an execution path reaching the string literal and evantually passing it to an output-generating statement (e.g., `echo` or `print`).

We approximate output candidates as those string literals containing 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 the entire corpus. 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, 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 distinctive classifier.

4.2 Experiment Setup

For the evaluation we symbolically executed a selection of PHP systems. We selected a corpus of twelve PHP systems with regard to system size and recency as the corpus of the case study for [10] only contained small-scale systems that are not maintained any more. The full list of PHP systems is shown in Table 1. Our selection of PHP systems includes

- four small-scale systems that selected from the previously mentioned case study in order to compare our results with the previous symbolic execution engine [10],
- three recent large-scale systems selected from the case study corpus of [7], and
- four small-scale systems selected from a list of recent Content Management Systems [1].

As for for the last two cases, we limited our selection since the parser we used did not support all language features used. For the experimental evaluation we symbolically executed per system each script file or entry point (files with a `.php`, `.inc`, `.bit` or `.module` extension). All following measurements are cumulated per system for each entry point.

4.3 Measuring Approximation Accuracy

In Section 4.1 we introduced the definition of output candidates as expected output. As a most accurate approximation contains all output candidates, we define two metrics to measure accuracy of our approximation.

First, 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. Second, 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. For an imprecise approximation, we may see loss of output candidate information resulting in a output coverage lower than the reach coverage.

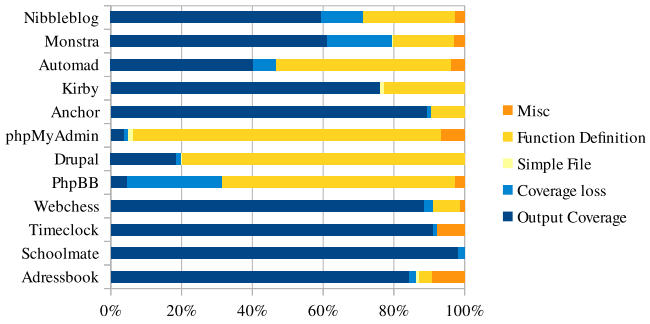

Figure 3: Results for output candidates: *Coverage loss* repesents reached output candidates that were not part of the output. *Simple file* and *function definition* represent context classifications of missed output candidates.

The coverage results are illustrated in Figure 3. 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. Of special interest are 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 (of course, unless PHP scripts are embedded) 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 3 illustrate, for almost all systems (except for AddressBook and TimeClock) not-reached output candidates had a function context. Note that SchoolMate was excluded from the diagram since our analysis

System	Version	Classification	SLOC	#files	#OCs	#includes
AddressBook	8.2.5.2	239	51,907	Adress Administration	XXX	XXX
SchoolMate	1.5.4	65	8,118	School Administration	XXX	XXX
TimeClock	1.04	63	20,800	XXX	XXX	XXX
WebChess	1.0.0	28	5,219	XXX	XXX	XXX
Drupal	7.5.0	125	52,464	Content Management System	XXX	XXX
phpBB	3.1.9	1,398	327,371	Bulletin Board	XXX	XXX
phpMyAdmin	4.6.3	871	303,582	Database Administration	XXX	XXX
Anchor	0.12.1	201	15,054	Content Management System	XXX	XXX
Kirby	XXX	XXX	XXX	Content Management System	XXX	XXX
Automad	XXX	XXX	XXX	Content Management System	XXX	XXX
Monstra	XXX	XXX	XXX	Content Management System	XXX	XXX
Nibbleblog	XXX	XXX	XXX	Content Management System	XXX	XXX

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

reached all output candidates for that system.

4.4.2 *Inaccessible Dynamic Features*

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 function can either be dead code if there is no call site for this function, or the function call failed. In PHP there are several ways to call a function beside direct call sites. The language offers indirect call mechanisms like callback-commands or an evaluation function that parses and evaluates PHP source code as strings.

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.

4.4.3 *Why did includes fail?*

To better understand include expression evaluation, we use another metric to approach accuracy from a different angle: We measure (1) the ratio of reached include expressions and the total number of include expressions in a system respectively (coverage of include expressions), and (2) the ratio of successfully resolved include expressions and reached include expressions in a system (resolution success rate). For a most accurate result all include expressions are reached and resolved.

For include expressions we sampled failed attempts and manually evaluated the cause of failure. We identified for (1) that all failed attempts were dynamic include expressions, i.e., those which were resolved at runtime and were assembled from fragments of environment-dependent information including database retrievals, user input, or files. Consequently, the resolved include expressions contained symbolic values which did not represent identifiable script files.

Finally, we measured for include expressions the overall reach coverage and the overall resolution success rate per system. As illustrated in Figure 4, except for the small-scale systems and two recent small-scale systems we did not even really reach a great portion of include expressions, although for most systems the resolution success rate was over 80 percent. Nevertheless, one failed attempt to include a script file can result in a cascade of missing more include

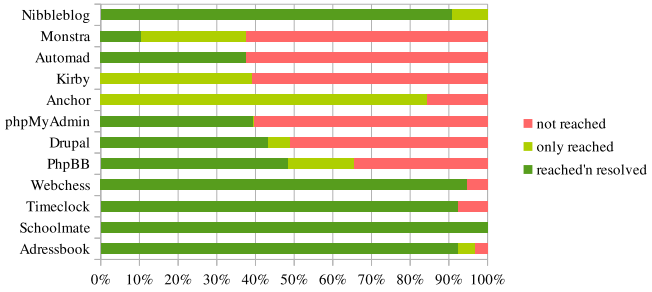

Figure 4: Include coverage and accuracy.

expressions as the include target script is not executed.

4.4.4 *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 5 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 5 function X can only be accessed through callback candidates, as for Figure 5 function Y can be partially accessed by callback candidates, whereas function Z is never accessible through callback candidates.

Like for include expressions, we again sampled dead function cases and identified the lack of direct call sites to those functions. As described in Section 4.4.2, functions despite having no direct call

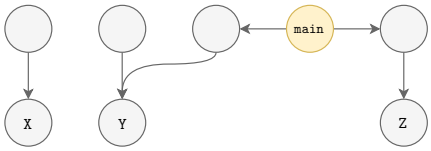

Figure 5: Callgraph analysis: The bottom nodes represent functions containing missed output candidates, the top nodes depict entry points to them.

sites were called through indirect call features. We identified dynamically assembled expressions (function names) to be responsible for failed attempts of calling a function indirectly as the resolved function name eventually contained symbolic values. These expressions, as well as include expressions, contained information dependent on the system environment.
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 4.4.2). As Figure 6 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.

5. LESSONS LEARNED

6. RELATED WORK

- Hills et al. [7]
- Output approximation using context-free grammar construction; applications [9, 14]

7. 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] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, Apr. 1978.

[7] M. Hills, P. Klint, and J. Vinju. 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, pages 325–335, New York, NY, USA, 2013. ACM.

[8] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[9] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web*, pages 432–441. ACM, 2005.

[10] H. V. Nguyen, C. Kästner, and T. N. Nguyen. 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, pages 518–529, New York, NY, USA, 2014. ACM.

[11] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Cross-language program slicing for dynamic web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 369–380, New York, NY, USA, 2015. ACM.

[12] H. V. Nguyen, C. Kästner, and T. N. Nguyen. 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, pages 693–696, Piscataway, NJ, USA, 2015. IEEE Press.

[13] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. 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, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[14] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

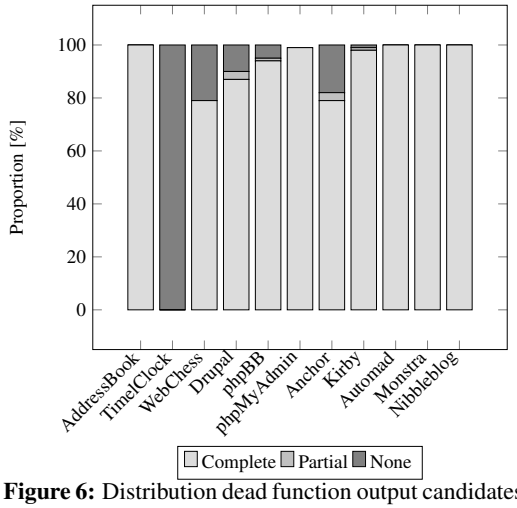

Figure 6: Distribution dead function output candidates that are partially or completely explained due to callback candidates.