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 at Dallas, 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 due to both conceptual limitations and unsupported language features.

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

Dynamic web applications have become widely popular and various different implementation techniques have emerged, ranging from server-side scripting languages to state-of-the-art frameworks for web development. For all ways to implement web applications, one common ground is the dynamic nature of output generation: HTML output is generated dynamically during a session, for instance when a form is submitted to a web application. Output can be dependent on different classes of inputs.

Symbolic execution is a static analysis technique between program testing and program proving. For a program, symbolic, i.e., abstract, yet fix inputs are applied instead of concrete inputs [2, 5]. 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. Symbolic execution extends concrete execution semantics with semantics to handle symbolic values. Hence, concrete execution semantics is contained in symbolic execution semantics.

Symbolic execution has been used and studied for generating test suites, but has more recently also been applied for approximating the output of web applications. Knowledge about different output variants enabled useful analyses for the domain of web applications, such as detecting and locating HTML validation errors [10]. As for developers, from an output model, outlines and callgraphs for easier IDE code navigation [7], or program slices [8] can be computed. Existing work has been integrated in the Varis plugin for the Eclipse IDE [9]. Despite the various use cases for analyses based on an approximated output model, static output approximation for PHP web applications has so far only been tested for smaller systems that are not developed anymore.

Can we have a practical tool?

We ask for the symbolic execution engine whether we can have a practical tool built from it. In the context of useful analyses stated earlier, we define a practical symbolic engine to be accurate and precise enough for the analyses' results to be useful. An accurate symbolic execution engine allows a single concrete execution to be retraced from a symbolic execution representing variability. For a precise symbolic execution engine, each concrete execution represented in a symbolic execution could have been performed by a concrete execution engine.

We see concerns for practicality in intrinsic properties of symbolic execution (abstraction of real input sources), and in the language properties of PHP. With no standardization the language has become successful one the one hand, but did not grew coordinately leaving constructs that are partially functionally redundant, or inconsistent for different versions.

We can have a practical tool, can't we?

To investigate this question, we build a symbolic execution engine for PHP that re-implements the specifications of previous symbolic execution semantics [7] for PHP with additional features, such as object-oriented programming, and extensive array support.

Based on our observations and evaluation results, we learn that for dynamic PHP constructs, expressions that are assembled dynamically, in combination with symbolic values symbolic execu-

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

DOI: 10.475/123_4

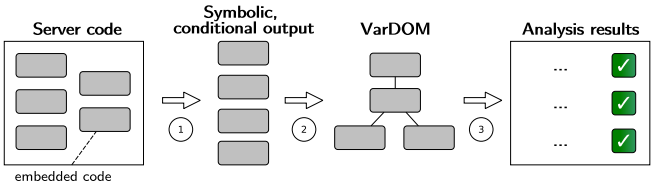


Figure 1: Approach overview; the different analyses in Varis are based upon the symbolic output. Step 1 is the *symbolic execution*, step 2 is the *variability-aware parsing* and step enfolds subsequent analyses based upon the symbolic output.

tion 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) Ihe tool infrastructure to statically approximate the output for PHP web applications using symbolic execution and (2) observations as well as an anecdotal and empirical evaluation to identify and measure PHP constructs impeding symbolic execution of PHP code.

2. STATE OF THE ART

2.1 Symbolic Execution (and Choices?)

2.2 Use Cases for Output Approximation

- Steps: Symbolic Execution → Parsing → Analysis, see Figure 1
- Varis: IDE support [9] for PHP with code slicing [8], code navigation [7] and HTML validation [10].

3. TOWARDS SCALABLE SYMBOLIC EXECUTION

3.1 Can we have a practical tool?

- Are the assumprions just engineering or do not matter for accuracy?

3.2 Oak: A Symbolic Interpreter

- OAK[1]: Re-implementing and extending the symbolic execution engine with OOP and array support
- Test-driven implementation strategy
- WordPress!

3.3 Experience Report

3.3.1 Problems

3.3.2 Concrete/Symbolic Boundary

[1] https://github.com/smba/oak

3.3.3 Further experiments

- SPL approach for Oak in order to evaluate different execution modes (feature model?)
- "*Things we tried, some improvement but had no significant impact; tradeoffs between accuracy and performance etc.*"

4. HOW CAN WE EXPLAIN LOW CODE COVERAGE?

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

4.1 How to measure approximation success?

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 the aproximation. We define a string literal to be an *output candidate* if there exists an execution path reaching the string literal and 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. We evaluated this heuristic manually with a sample of 400 string literals from the entire corpus and measured a precision of 94% and a recall of 50%. This means, our classification has a rate of 6% false positives and only half of the actual output candidates respond. These were mostly HTML-containing literals, whereas plain text did not respond.

4.2 Experimental Evaluation

For the evaluation we chose to selected a corpus of twelve PHP systems with regard to previous work, but chose to additionally scale system size and recency. The full list of twelve PHP systems is shown in Table 1. Our selection contains twelve systems:

- Four small-scale/old systems are systems that have been subject to related studies and approaches in order to have comparable results. These projects do not make comprehensive use of PHP language features and output approximation on these systems achieved fairly high precision [6, 7].
- Three large-scale/modern systems that have been subject to related studies, especially their use of language features as we aim to study scalability of symbolic execution The feature usage of these systems has been studied by Hills et al. [3]
- Five small-scale/modern systems to compare the impact of system size to the second part of the corpus. The systems are selected from a list of recent content management systems (CMS) [1]. We only selected five systems due to limited support of the Quercus parser used in our implementation.

For the experimental analysis 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 Metrics for Understanding Limitations

The following metrics are used throughout the description of the results.

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

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

4.3.1 Reach Coverage and Output Coverage

In Section 4.1 we introduced the definition of output candidates. In order to describe how much of the expected output was actually reached by the symbolic interpreter or even was part of the output respectively. We define the metric *reach coverage* as the ratio of output candidates that are reached by any execution path during symbolic execution and the total number of output candidates in the analyzed system. Moreover, the metric *output coverage* describes the ratio of output candidates that are contained in the output of the symbolic interpreter and the total number of output candidates of the analyzed system.

4.3.2 Include Expressions: Coverage and Success

One metric to approach *accuracy* is to measure (1) the ratio of reached include expressions and the total number of include expressionsm in a system respectively, and (2) the ratio of successfully resolved include expressions and reached include expressions in a system. We therefore measure the coverage of include expressions as well as the resolution success rate. For a most accurate result, all include expressions are reached and resolved.

4.3.3 String Literal Context

Another approach to understand not-reached output candidates is to understand why their surrounding program code was not accessed/accessible. Of special interest are those output candidates which are located in HTML template files or are surrounded by a function definition.

First, HTML templates are script files containing HTML and additional nested script snippets. Any HTML inside a template is part of the output once the template file is included. Consequently a not-reached output candidate residing in a template file means that the file is not included (dead code) or an include failed at runtime. Second, an output candidate located inside a function definition indicates that either the defined function is never called (dead code), or that the entire function definition is never reached, hence, it is not included or has failed to be included.

Apparently the explanations for both two cases are not mutually exclusive as the surrounding code does not give any reference to why it has not been accessed or could not be accessed. Nevertheless, this classification of not-reached output candidates may give us some insight on the question whether code accessibility is a matter, and if so, where to look for further answers. We will refer to this two cases as *function context* and *template context*.

4.3.4 Dead Code Features

A refinement of the previous metric is to explicitly measure those cases where (1) not-reached output candidates are located inside a function context and the corresponding function is defined at runtime. Here we can a priori exclude a failed include from the set of possible explanations. We are also interested in those cases where (2) the not-reached output candidate is located in a file that has never been included from any other script file at runtime). Consequently these script files are either never to be included, or every attempt to include these files failed.

We refer to those two cases as dead code features, especially (1) *dead functions* or (2) *dead includes*. The explanations for these two cases are mutually exclusive since for (1) relevant script files need to be included and for (2) relevant script files must not be included.

4.3.5 Callback Candidates

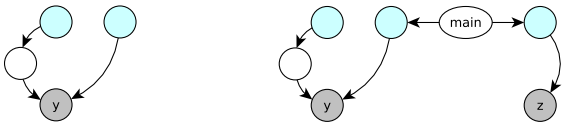
Aside from failed or successful includes, this metric approaches the explanation of why functions containing not-reached output candidates are never called. The trivial explanation would be that there are no direct calls of those functions, yet PHP offers a number of ways to call a function indirectly, for instance using built-in functions (`call_user_func` or `call_user_func_args`), 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 2 illustrate the idea of detecting the functions' access points: The green function definitions depict callback candidates, the gray ones depict function definitions containing not-reached output candidates. In Figure 2a function `x()` can only be accessed through callback candidates, as for Figure 2b function `y()` can be partially accessed by callback candidates, whereas function `z()` is never accessible through callback candidates.

4.4 Results



(a) Completely explained by callback candidates (no call from `main`)

(b) Partially (left; one call from `main`) or not at all (right) explained by callback candidates

Figure 2: Callgraph analysis. The blue nodes represent entry points to relevant functions (gray).

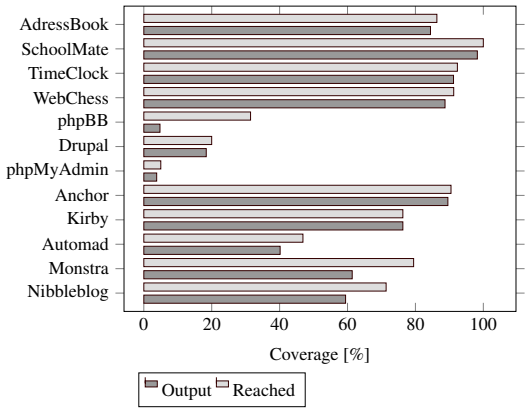


Figure 3: Coverage results for the PHP corpus

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 % respectively. For the three modern/large-scale systems we measured poor coverage with reach coverage ranging from 5 % to 30 % and output coverage ranging from 4 % to 20 %. We measured medium to high coverage for the more recent small-scale systems: reach coverage ranging from 43 % to 85 %, output coverage ranging from 40 % to 85 %.

4.4.1 *Output Candidates in Function Definitions*

Based on these results we investigated why output candidates could not be reached. First, we started classifying not-reached output candidates by their string literal context (see Section 4.3.3). As the classification results in Figure 4 illustrate, for almost all (except for AddressBook and TimeClock) systems not-reached output candidates had a function context. Note that SchoolMate was excluded from the diagram since our analysis reached all output candidates for that system.

4.4.2 *Developing Hypotheses: Inaccessible Code*

Furthermore, this lead us to investigate whether we could explain not-reached output candidates by inaccessible dead code features (see Section 4.3.4), especially either dead functions or dead includes. As Figure 5 illustrates, the two explanation hypotheses did explain a portion of not-reached output candidates completely. If we look at Anchor for example, we see that all not-reached output candidates had a function context, and that 30 % were explained by dead functions and dead includes respectively. The missing 40 % correspond to functions that were defined and called, but whose function body was not completely covered by the execution, i.e., certain branches were missed. Nevertheless, we see that inaccessible code is possibly one of the main reasons for missing output candidates.

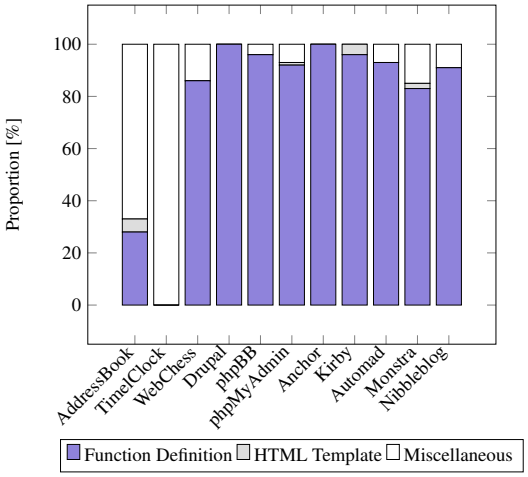


Figure 4: Context distribution of never reached output candidates for the PHP corpus.

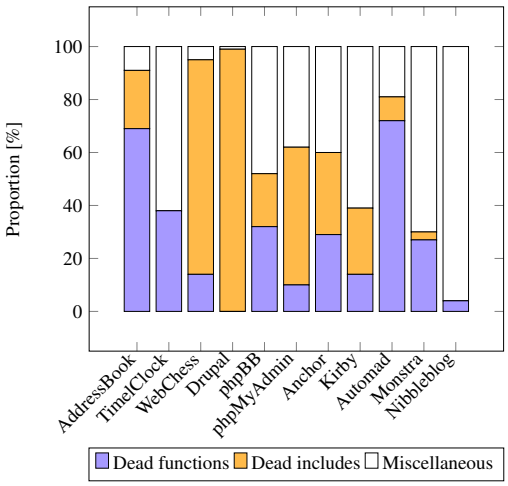


Figure 5: Explanation of never reached output candidates by dead code features.

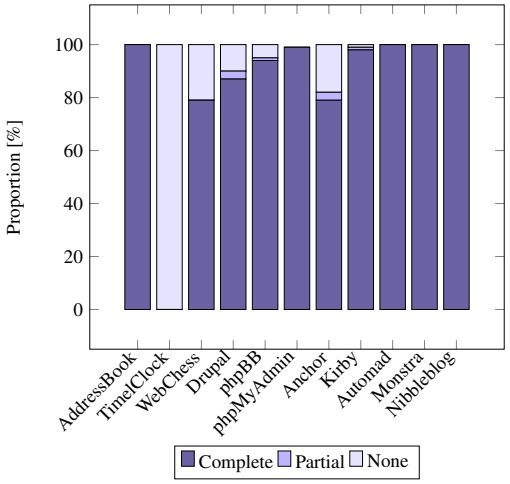


Figure 7: Distribution dead function output candidates that are partially or completely explained due to callback candidates.

4.4.3 *Why were code sections inaccessible?*

Eventually we started analyzing (1) why include expression may not be resolved/evaluated correctly, and (2) why a reasonable portion of defined functions is never called. 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 are 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 apparently did not represent identifiable script files.

For (2) we again sampled dead function cases and identified the lack of direct calls to those functions. Interestingly, also functions despite having no direct call sites were called through indirect call features (see Section 4.3.5). Again 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.

4.4.4 *Quantitative Analysis*

Finally, we measured for include expressions the overall reach coverage and the overall resolution success rate per system. As illustrated in Figure 6a, 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 % (see Figure 6b). As described in Section 4.3.2 one failed attempt to include a script file possibly results in a cascade of missing more include expressions.

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 4.3.5). As Figure 7 illustrates, around 80 % (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.

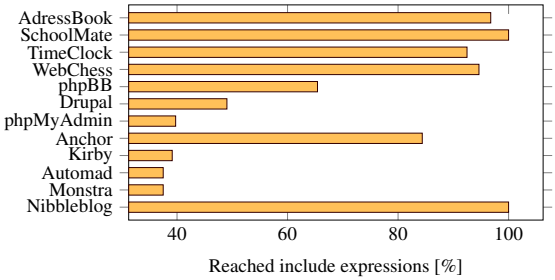
5. LESSONS LEARNED

- Certain language features are prone to be used in a way that symbolic execution is impeded.
- Tendency to use functions/includes with I/O, databases or configuration files is problematic.

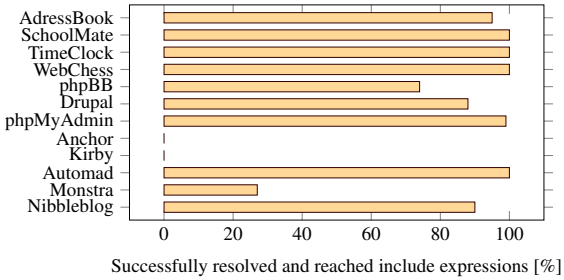
> The use of include expressions whose evaluation at runtime involves concatenation, especially of fragments from critical sources, impede symbolic execution due to inaccuracy.

6. RELATED WORK

- Static output approximation using context-free grammars by Minamide [6] and detection of SQL injection vulnerabilities [12].
- Static analysis of feature usage in PHP systems by Hills et al. [3] and static resolution of include expressions [3, 4]
- bener [5] [11]



(a) Proportion of reached include expressions per project.



(b) Rate of successful resolutions of reached include expressions per project.

Figure 6: Include coverage and accuracy.

7. REFERENCES

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

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

[3] 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.

[4] M. Hills, P. Klint, and J. J. Vinju. Static, lightweight includes resolution for php. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 503–514. ACM, 2014.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] K. Sen, G. Necula, L. Gong, and W. Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853, New York, NY, USA, 2015. ACM.

[12] 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.