# Supporting PHP Dynamic Analysis in PHP AiR

Mark Hills
East Carolina University, Greenville, NC, USA
mhills@cs.ecu.edu

## Abstract

The PHP AiR framework is currently being developed to support software metrics, empirical software engineering, and program analysis for real-world PHP systems. While most of the work on program analysis has focused on static analysis, to help address the dynamic nature of the language we have also started to extend PHP AiR with support for dynamic program analysis. This extended abstract highlights two parts of this support: integration with xdebug for trace analysis, and instrumentation of an open-source PHP interpreter with a focus on supporting string origins, allowing us to explore how strings are created in security-sensitive areas such as database calls and HTML generation.

***Categories and Subject Descriptors*** F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

***General Terms*** Languages

***Keywords*** Dynamic analysis; dynamic language features; PHP

## 1. Introduction

The PHP Analysis in Rascal (PHP AiR) framework [7] is currently being developed in Rascal [11] as an extensible program analysis framework for PHP. The work on this framework has focused on three areas: software metrics, empirical software engineering, and program analysis. To make sure PHP AiR is useful in real-world scenarios, we have mainly targeted large open-source frameworks and applications such as Symfony (one of the most popular MVC frameworks), CakePHP (another popular MVC framework), WordPress (a blogging and CMS system), and MediaWiki (the wiki software used for Wikipedia). Published research using PHP AiR has focused on empirical studies of open-source software and on program analysis, including a study of PHP feature usage [8]; a study of how the use of dynamic language features has evolved in specific applications [5]; a static analysis for resolving dynamic file includes in PHP scripts to a set of one or more files that could be included at runtime [9]; and a study of how variable features—which allow identifiers to be given by expressions that evaluate to strings—are used in PHP scripts, breaking these uses down into a number of common patterns that can be statically resolved to the names that will be used at runtime using lightweight program analysis techniques [6].

While most of our work on program analysis has focused on static analysis, to help address the dynamic nature of the language we have also started to extend PHP AiR with support for dynamic program analysis. This extended abstract highlights two parts of this support. First, we are extending PHP AiR to work with execution traces generated by xdebug[1], a popular debugging and tracing tool for PHP. Second, we are instrumenting an open-source PHP interpreter to support string origins [10], allowing PHP AiR analysis scripts to explore how strings are created in security-sensitive areas such as database calls and HTML generation. The PHP AiR framework is available online;[2] we expect to release initial versions of the extensions discussed here soon.

## 2. Execution Tracing with xdebug

xdebug provides support for generating execution traces, called function traces in the documentation, which are saved to trace files with a well- defined format generated as the PHP code on the site is executed. These traces provide information about which functions are called, which actual parameters are passed, which values are returned, and (optionally) variable assignments that occur during execution of a function. The debugger can be used to provide even more detailed information at a specific point of execution in the program, such as the call stack and the values assigned to specific variables and object properties.

Our initial motivation for gathering execution traces was to collect information on how *dynamic invocations*[3] are used in plugin-based systems, such as in WordPress, and (if possible) to then transform these dynamic invocations into regular function calls, thus allowing more accurate static analysis (e.g., call graph construction) and more opportunities for compilers (e.g., PHC [2] or the HipHop compiler [18]) to optimize website performance. Using profiling information from running functional tests of the site, we transformed instances of dynamic invocations in WordPress to invocations of the functions that were actually selected at runtime, using conditionals to select the proper function call based on runtime values [13]. Since testing could miss some possible invocations, to maintain the soundness of this transformation an option to add a default case, using the original dynamic invocation, was also provided.

This work is similar to the work of Furr et al. [3], who used profiling of dynamic Ruby features, in conjunction with existing test cases, to determine how the dynamic features of a program are used in practice. Similarly finding that many of these features were used in ways that were almost static, they were able to replace them with static versions that were then amenable to static type inference in a system such as DRuby [4]. The transformation they defined was sound, with a default path still specified for unprofiled

[1] `http://xdebug.org/`

[2] `https://github.com/cwi-swat/php-analysis`

[3] Dynamic invocations allow a function or method to be invoked indirectly, given the function or method name and the actual arguments.

dynamic cases but using a custom variant to properly deal with type errors. Unlike in their work, in our case we did not find adequate test cases, with some dynamic features receiving minimal coverage and some features not executed by any test case, requiring us to create a number of new functional tests.

To continue this work, which made use of external tools such as sed scripts, we are extending our support for processing these traces directly in PHP AiR using a custom PHP trace grammar and user-defined filters and trace actions (given in Rascal as closures) to decide which events to keep and how to react to them as they are encountered. We have also implemented a programmable xdebug client in Rascal that communicates with xdebug using TCP/IP sockets and the xdebug DBGp protocol.[4] This allows us to script interactions with the debugger directly using Rascal scripts. Longer term, we plan to investigate techniques related to directed symbolic execution [12] to generate tests that will exercise specific program locations, allowing us to augment the test cases provided with systems such as WordPress. We also plan to investigate what kinds of performance improvements, if any, we see for the transformed code when using PHP compilers and runtimes such as HHVM.

## 3. Integration with Quercus

The Quercus PHP server[5] is shipped as part of Caucho Resin,[6] a high performance J2EE server. Written in Java, Quercus supports PHP 5 and a number of standard PHP extensions, such as the MySQL extensions used to access MySQL databases, and has been tested with a number of popular PHP applications including WordPress and MediaWiki. The code in the interpreter follows a logical structure, making it easy to find and instrument the classes that manage specific types of values (e.g., PHP strings) or represent specific language constructs (e.g., string concatenation).

As initial work, we have begun extending Quercus to support string origins [10], a technique related to origin tracking [17]. Origin tracking was introduced in the context of term rewriting systems to map fragments of intermediate terms back to the initial term. String origins were similarly introduced as a lightweight method for tracing the effects of program transformations. In string origins, Rascal source locations, given as URIs augmented with the starting offset and length of a text block in a source file, are used to indicate the origins of program fragments. Transformation steps then maintain this origin information as specific transformation operations are applied, making it possible to tell that certain parts of a target program are based on specific input text or were synthesized as part of a specific transformation step.

Our interest in string origins is similar, but instead of program transformation we are interested in program evaluation. To support string origins, we are extending the Quercus classes representing string values and string operations to track origin information, allowing us to track how strings are provided and manipulated at runtime. Given specific points of interest, such as database calls, statements that generate HTML, or dynamic language features, we can then explore where the strings that reach these points come from, including strings provided as user inputs, read from database queries, or written to specific configuration files.

This information will feed back into our work on tracing, described above. For instance, a dynamic feature that is based directly on strings input by the user (e.g., code in an administrative console that allows arbitrary functions to be called or expressions to be executed) could not be converted soundly into static variants without providing a default case, and the behavior may be varied enough that even trying to generate static variants would not be worthwhile. This will also be important for empirical research and program analysis targeting database calls and HTML generation. In earlier work we looked at refactoring database calls to use prepared statements instead of queries constructed using string building operations [16]. This work categorized queries into an initial collection of patterns. Better knowledge of how these queries are built in a varied corpus of systems would lead to a better classification and potentially more precise analysis and transformation algorithms.

There is a wide variety of work on program tracing and instrumentation. Similar work in the context of PHP includes the Apollo system [1], which included a so-called *shadow interpreter* to record information needed for the analysis. Similar work on analysis related to database use includes recent work on using SQL execution traces for program comprehension [15] and on finding the location of specific SQL queries in Java source [14].

[4] `http://xdebug.org/docs-dbgp.php`

[5] `http://quercus.caucho.com/`

[6] `http://caucho.com/`

## References


[1] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Dynamic Web Applications. In *Proceedings of ISSTA 2008*, pages 261–272. ACM, 2008.

[2] P. Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin, April 2010.

[3] M. Furr, J. hoon (David) An, and J. S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Proceedings of OOPSLA 2009*, pages 283–300. ACM, 2009.

[4] M. Furr, J. hoon (David) An, J. S. Foster, and M. W. Hicks. Static Type Inference for Ruby. In *Proceedings of SAC 2009*, pages 1859–1866. ACM, 2009.

[5] M. Hills. Evolution of Dynamic Feature Usage in PHP. In *Proceedings of SANER 2015*, pages 525–529. IEEE, 2015.

[6] M. Hills. Variable Feature Usage Patterns in PHP. In *Proceedings of ASE 2015*, 2015. To Appear.

[7] M. Hills and P. Klint. PHP AiR: Analyzing PHP systems with Rascal. In *Proceedings of CSMR-WCRE 2014*, pages 454–457. IEEE, 2014.

[8] M. Hills, P. Klint, and J. J. Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of ISSTA 2013*, pages 325–335. ACM, 2013.

[9] M. Hills, P. Klint, and J. J. Vinju. Static, Lightweight Includes Resolution for PHP. In *Proceedings of ASE 2014*, pages 503–514. ACM, 2014.

[10] P. Inostroza, T. van der Storm, and S. Erdweg. Tracing Program Transformations with String Origins. In *Proceedings of ICMT 2014*, volume 8568 of *LNCS*, pages 154–169. Springer, 2014.

[11] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-Proceedings of GTTSE 2009*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[12] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed Symbolic Execution. In *Proceedings of SAS 2011*, volume 6887 of *LNCS*, pages 95–111. Springer, 2011.

[13] C. Mulder. Reducing Dynamic Feature Usage in PHP Code. Master's thesis, University of Amsterdam, 2013.

[14] C. Nagy, L. Meurice, and A. Cleve. Where Was This SQL Query Executed? A Static Concept Location Approach. In *Proceedings of SANER 2015*, pages 580–584. IEEE, 2015.

[15] N. Noughi and A. Cleve. Conceptual Interpretation of SQL Execution Traces for Program Comprehension. In *Proceedings of PCODA 2015*, pages 19–24. IEEE, 2015.

[16] I. Rucareanu. PHP: Securing Against SQL Injection. Master's thesis, University of Amsterdam, 2013.

[17] A. van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.

[18] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans, and S. Tu. The HipHop Compiler for PHP. In *Proceedings of OOPSLA 2012*, pages 575–586. ACM, 2012.