Query Construction Patterns in PHP

David Anderson, Mark Hills
East Carolina University, Greenville, NC, USA
andersonda12@students.ecu.edu, mhills@cs.ecu.edu

***Abstract*—Most PHP applications use databases, with developers including both static queries, given directly in the code, and dynamic queries, which are based on a mixture of static text, computed values, and user input. In this paper, we focus specifically on how developers create queries that are then used with the original MySQL API library. Based on a collection of open-source PHP applications, our initial results show that many of these queries are created according to a small collection of query construction patterns. We believe that identifying these patterns provides a solid base for program analysis, comprehension, and transformation tools that need to reason about database queries, including tools to support renovating existing PHP code to support safer, more modern database access APIs.**

I. Introduction

PHP, originally created for building simple dynamic webpages, is now one of the most popular programming languages. As of November 2016, it ranks 7th on the TIOBE index,[1] which measures interest in various programming languages, and is used by 82.3 percent of all websites whose server-side language can be determined.[2] PHP is the fourth most popular language by number of GitHub repositories.[3]

PHP was designed to allow the rapid construction of websites, and includes a number of libraries for interacting with databases. One of the most common libraries used for database access is the original MySQL API. This library, available since version 2 of PHP, provides a procedural interface for querying MySQL databases. Deprecated in version 5.5.0 of PHP, this library is no longer supported in the newest version of the language, version 7. Newer applications should instead use libraries such as MySQLi (MySQL Improved) or PDO (PHP Data Objects). These libraries provide an object-oriented API and include support for additional features, such as prepared statements, stored procedures, and transactions. Given the amount of code using the original API, it is important to provide automated support for identifying existing API calls and the related queries, allowing them to be transformed to safer variants that take advantage of newer features—such as prepared statements with parameter binding—in newer APIs.

One immediate challenge is that PHP includes a number of dynamic features, some of which are commonly used [1], [2], making it challenging to statically reason about PHP code. In this paper, we present initial work on identifying patterns used by developers to construct MySQL queries in PHP code, which can then be leveraged to inform program analysis and transformation tools. These patterns include static occurrences, such as the direct use of string literals; occurrences that differ based on control flow, such as cases where different query strings are used based on conditionals; queries that are dynamic just in the values of supplied parameters, such as those used in a `SET` or a `WHERE` clause; and queries that are truly dynamic, with the query body itself (not parameter values) based on user inputs or language features such as function and method calls. Using an initial corpus of 10 open-source systems, our initial results show that many queries are static or are only dynamic in parameters, with very few truly dynamic queries. We believe this information will be useful in our ongoing work on transforming PHP code to use newer, more secure database APIs, allowing us to initially focus on building precise support for the most common query patterns.

The rest of the paper is organized as follows. In Section II we describe the corpus used in this paper, while Section III provides details on how the study was conducted. Section IV then describes the query construction patterns we have identified so far, categorizing how queries are actually built in PHP code, with Section V showing our initial results on how often these patterns actually arise in our corpus. We then briefly discuss related work in Section VI, while Section VII discusses future work and concludes. All software used in this paper is available for download at https://github.com/ecu-pase-lab/mysql-query-construction-analysis.

II. Corpus

10 systems that use the MySQL API were selected for the initial corpus. Four of these systems were chosen based on their use in earlier experiments [3], [4]. This includes Schoolmate 1.5.4, a tool for school administration; WebChess 0.9.0, an online chess game; FAQ Forge 1.3.2, a tool for creating and managing FAQs and other documents with hierarchical structures; and geccbblite 0.1, a simple bulletin board application. These are all older systems, and are no longer maintained. The other six systems are a mix of older and more recent systems. This includes MyPHPSchool 0.3.1, a content-management system for primary and secondary schools (US Kindergarten through grade 12); Fire-Soft-Board 2.0.5 and UseBB 1.0.16, both bulletin-board systems; OpenClinic 0.8.2, a medical records system; OMS 1.0.1, an organization management system focused on school groups; and web2project 3.3, an online project management system. Most of these systems are available on either SourceForge (Schoolmate, WebChess, FAQ Forge, geccbblite, MyPHPSchool, OMS) or GitHub (Fire-Soft-Board, OpenClinic, web2project), with one, UseBB, available directly from the UseBB homepage. For the

[1] http://www.tiobe.com/tiobe-index/
[2] https://w3techs.com/technologies/details/pl-php/all/all
[3] https://github.com/blog/2047-language-trends-on-github

978-1-5090-5501-2/17 © 2017 IEEE

SANER 2017, Klagenfurt, Austria
Early Research Achievements

TABLE I
THE CORPUS.

System	Version	File Count	SLOC
FAQ Forge	1.3.2	17	1040
Fire-Soft-Board	2.0.5	271	47464
geccBBlite	0.1	11	304
MyPHPSchool	0.3.1	70	8230
OMS	1.0.1	16	2234
OpenClinic	0.8.2	170	16610
Schoolmate	1.5.4	63	6554
UseBB	1.0.16	84	12650
web2project	3.3	584	100544
WebChess	0.9.0	24	3525

first four systems which were used in other experiments, the same version used there was also used here; for the other six, the most recent regular release (e.g., not a beta) was used.

Table I shows information on the corpus, including the system name, the version number, the number of PHP source files (including .php and .inc files), and the number of source lines of code, counted using the cloc[4] tool. In total, the corpus includes 10 systems, consisting of 199,155 lines of PHP code and 1,310 PHP source files.

III. RESEARCH METHOD

PHP AiR [5], [6], a framework for PHP Analysis in Rascal, is used to perform all the analysis used to identify the patterns described in Section IV and compute the results reported on in Section V. The PHP AiR framework is written in Rascal [7], a meta-programming language focused on program analysis and transformation, and makes heavy use of Rascal language features such as pattern matching, source locations (indicating regions in source code), and data types such as relations, lists, and maps. PHP AiR adds support for parsing PHP code, building control flow graphs and analyses over these graphs, and simplifying PHP expressions using rules about PHP constants, operations over strings, and commonly-used library functions. The analysis needed to generate all results that are reported in this paper, including the associated tables, is scripted, ensuring the results can easily be reproduced and checked. The PHP AiR framework is available on GitHub at https://github.com/cwi-swat/php-analysis, while all scripts used in this paper are also available on GitHub at https://github.com/ecu-pase-lab/mysql-query-construction-analysis.

Section II described how the studied version of each system in the corpus was selected. The code for each was downloaded from SourceForge, GitHub, or the project homepage, depending on the system, and then parsed without modification using PHP AiR. The analysis code detects patterns using a combination of pattern matching over ASTs, regular expression matching, and analysis performed over the control flow graphs constructed for each PHP script. Details on how each pattern is detected are included with the descriptions of the patterns in Section IV. Tables reporting these results are generated directly based on the counts of these detected patterns.

[4]https://github.com/AlDanial/cloc

IV. QUERY CONSTRUCTION PATTERNS

To support precise program analysis and transformation tools for PHP code that uses databases, it is important to understand how the database queries used by database libraries are typically constructed. In this section, we describe an initial set of Query Construction Patterns (QCPs), used to recognize common idioms for building queries, as well as the PHP AiR analysis used to recognize each pattern. Brief code examples from the corpus are given for each.

A. QCP1: Literal Query Strings

QCP1 detects cases where the query used in a call to `mysql_query`, the main query function in the original MySQL API, is given as a string literal, or as a string expression that can be transformed into a string literal using the expression simplification logic mentioned in Section III. QCP1 appears in two variants. In the first, a string literal is passed directly to `mysql_query`, as in this code snippet from line 174 of *ManageStudents.php* in Schoolmate:

```
$query = mysql_query("SELECT termid, title FROM terms");
```

In the second, the literal is assigned to a variable, with this variable then used in the call to `mysql_query`, as in this snippet that starts on line 30 of *lib/edit-page.inc* in FAQ Forge:

```
$q = "SELECT id FROM Faq WHERE parent_id = 0 ORDER BY id";
$result0 = mysql_query ($q, $dbLink);
```

For the first case, this pattern is recognized directly using pattern matching over ASTs, identifying calls to `mysql_query` that are given a string literal for the query argument. The second case uses a reachability analysis over the control flow graph (CFG) for the function, method, or script containing the call. This analysis checks to ensure that the same literal is assigned to the variable on all paths that reach the call.

B. QCP2: Cascading Concatenating Assignments

QCP2 detects cases where the query is built over several lines of code using a series of concatenating assignments (i.e., assignments using `.=` instead of =). The following, from line 41 of *lib/view-doc.inc* in FAQ Forge, is a simple example of QCP2, with only a single concatenating assignment:[5]

```
$q  = "SELECT * FROM Faq WHERE parent_id = 0 AND publish =
      'y' ";
$q .= "ORDER BY list_order";
$result0 = mysql_query ($q, $dbLink);
```

To identify this pattern, pattern matching is used to detect `mysql_query` calls with a variable parameter, to find initial assignments into this variable, and to find sequences of concatenating assignments after each initial assignment. A forward analysis is then performed from each assignment to match it to reachable `mysql_query` calls, with checks for other assignments into the variable holding the query that would kill the assignment on that path.

[5]The most common number of concatenating assignments used to build a query in the corpus is just 1, while the largest number is 8.

C. QCP3: Assignments Distributed over Control Flow

QCP3 detects cases where the query parameter to `mysql_query` is a variable and where that variable can take on multiple possible assignments (distinct query strings) based on control flow. For this pattern, we identified two sub-cases:

- QCP3a: Assignments of literals, distributed over control flow structures
- QCP3b: QCP4 assignments (see below), distributed over control flow structures

Identifying occurrences of both cases requires a reachability analysis. In QCP3a, the relevant CFG is referenced to determine if all paths that reach the `mysql_query` call include literal assignments to the query variable and if the number of reaching query expressions is greater than one. QCP3b is similar, but checks to see if assignments into the query variable use queries formed according to QCP4. The following example of QCP3a, which detects three variants for the query, starts on line 60 of file *html/resources.php* in MyPHPSchool:

```
$sql = "SELECT * FROM resources WHERE type='encyclopedia'
    ORDER BY name";
if($order_by == "votes"){
  $sql="SELECT * FROM resources WHERE type='encyclopedia'
       ORDER BY votes DESC";
} elseif ($order_by == "average"){
  $sql="SELECT * FROM resources WHERE type='encyclopedia'
       ORDER BY average DESC";
}
$result = mysql_query($sql, $db);  //Execute SQL
```

D. QCP4: Dynamic Query Strings

QCP4 detects cases where the query parameter to `mysql_query` is dynamic. We have identified three sub-cases, based on the type of expression or expressions used to construct the query string:

- QCP4a: An encapsed (interpolated) string containing a mixture of literal fragments, PHP variables, function calls, and possibly other non-literal expressions.
- QCP4b: A concatenation operation between string literals and variables or other non-literal expressions.
- QCP4c: A variable containing an expression built according to QCP4a or QCP4b.

The following example, of QCP4a, is from line 3 of *VisualizeClasses.php* in Schoolmate:

```
$query = mysql_query("SELECT title FROM semesters WHERE
    semesterid = $_POST[semester]");
```

The following example, of QCP4c, starts on line 40 of *lib/functions.inc* in FAQ Forge:

```
$q = "SELECT * FROM Faq WHERE parent_id = $id";
$result = mysql_query ($q, $dbLink);
```

For QCP4a and QCP4b, identifying the patterns only requires pattern matching over the ASTs. For the third case, a flow analysis over the relevant CFG is used to determine if assignments into the query variable are of query strings formed according to QCP4a and QCP4b.

To better understand how the dynamic portions of QCP4 queries are used, we have further analyzed QCP4 occurrences to answer the following two questions:

- Q1: What types of PHP constructs make up the dynamic portions of QCP4 occurrences (e.g. variables, function calls) and how many times do each of these types occur?
- Q2: In QCP4 occurrences, what parts of the query are static and what parts are dynamic?

Q1 is important since it provides additional details about which language features are used to build queries (and, by extension, which are used rarely or not at all). Q2 then tells us where these dynamic parts of queries are used—as parameters to constructs like `SET` or `WHERE`, or directly to form the body of the query (e.g., to provide names of tables, names of columns, or operators). To answer Q1, we use pattern matching over the ASTs for each dynamic part of a QCP4 query. Dynamic query parts are grouped based on their AST node type. For query parts that are retrieved by fetching an array element, we differentiate between user-defined arrays and PHP *superglobal* arrays such as `$_GET` and `$_POST`, which hold parameters passed as part of the URL or through a posted form, respectively. To answer Q2, we use regular expression matching to determine where static parts of queries end. For example, if a static query part ends with an operator and is followed by a dynamic query part, we can infer that the dynamic query part was used as a parameter to that operator. The results of this analysis are described in Section V.

V. Initial Results

Table II describes the number of occurrences in the corpus of each pattern described in Section IV. The corpus contains a total of 617 calls to `mysql_query`. Of these, 488 are variants of QCP4, accounting for roughly 79% of all `mysql_query` calls. The second most common pattern, QCP1, accounts for roughly 16% of all calls with 96 occurrences. The other patterns are much less common, while only 10 of the 617 queries in the corpus are unclassified by our analysis. Through manual examination of these unclassified queries, we determined that one uses a mix of multiple patterns along different program paths; one includes a possible path that reaches the `mysql_query` call with no assignment into the query variable used in the call; two are QCP4 occurrences that are stored in (and referenced from) an array rather than a variable; five use a function or method parameter as the variable containing the query; and one is a mixture of QCP2 and QCP3, where the query is built using a series of concatenating assignments distributed over conditionals.

TABLE II
Counts of Each Query Construction Pattern.

Query Construction Pattern	Number of Occurrences
QCP1	96
QCP2	17
QCP3a	2
QCP3b	4
QCP4a	198
QCP4b	85
QCP4c	205
Other	10

TABLE III
COUNTS OF EACH QCP4 DYNAMIC QUERY PART BY TYPE.

Type	Number of Occurrences
Function Call	24
Ternary Operator	2
Fetch Array Element	183
Fetch Superglobal Element	386
Variable	415

Exploring QCP4 in more depth, Table III shows how many times different PHP language features are used as dynamic parts of QCP4 query strings, addressing Q1 from Section IV. 97.4% of all dynamic query parts in the corpus come from array elements, superglobal elements, and variables. Function calls and uses of the ternary operator account for the rest. Table IV then presents the results of grouping dynamic query parts based on the part of the query they represent, addressing Q2 from Section IV. Because of the relatively complex grammar of SQL, our regular expressions are not able to categorize all dynamic parts of queries into a specific role. Upon manual inspection of the 11 queries that fall under the "Other" category, we were able to determine that 4 of these should be categorized as "Column, Table, or Database Name", while 5 should be categorized as "Parameter". For the remaining two cases, the first is a PHP encapsed string where the string consists of just a single variable, while the second includes a variable used to build a complex SQL clause using various assignments and control flow statements.

A. Threats to Validity

The main threat to validity is that the evaluation has used a corpus of just 10 systems, some of which are older and may use programming idioms that are no longer common (but may be common in other, older code that needs to be maintained). This is mitigated somewhat by the variety of systems used, but further evaluation is needed. Another threat is that the control-flow analysis is not sound, specifically in the case of where aliases of a variable holding the query text are taken. Based on manual examination of the code, we do not believe this effects the results reported in this paper, but a sound variant of the analysis will be needed to ensure the safety of any code transformations relying on the identified query patterns. Finally, given the complexity of the SQL grammar, the regular expression matches used to determine the roles for dynamic query fragments in QCP4 may mis-identify some roles. To mitigate this, we have validated the roles assigned to two groups of 100 fragments, chosen randomly, determining that all roles were assigned properly. This gives us confidence that the assignment of roles is accurate.

VI. RELATED WORK

A number of static and dynamic techniques have been used to analyze database interactions in programs. Cleve and Hainaut [8] used aspect-oriented programming to instrument Java JDBC classes, allowing traces with information on executed database queries to be generated to support application reverse engineering. Noughi et al. [9] used tracing to gather information on queries executed as part of performing specific application scenarios, with further analysis determining which parts of the schema were accessed and how sequences of queries were related. This was implemented in the DAViS [10] plugin to DB-Main, and included support for visualizing the analysis results. Alalfi et al. also used tracing in a system called WAFA [11]. WAFA uses the TXL programming language [12] to instrument web applications, in this case in the PHP language (although the technique itself is not specific to PHP). This instrumentation saves information on queries (and other elements of the application) into a database, and later matches this information to specific calls to the MySQL API. Query strings are initially detected by matching on SQL keywords such as `SELECT` and `INSERT`.

TABLE IV
COUNTS OF EACH QCP4 DYNAMIC QUERY PART BY ROLE.

Role	Number of Occurrences
Parameter	982
Column, Table, or Database Name	17
Other	11

Nagy et al. [13] created a static analysis to identify the set of program locations where a specific input query could be executed. These locations either contain explicit queries, such as through regular JDBC code, or implicit queries, triggered using higher-level APIs such as Hibernate. The analysis works by representing possible queries as patterns over the MySQL dialect of SQL, with variable parts of the query represented as *Joker* nodes that can match anything. The locations that could have executed the input query are then identified by matching the patterns defined for each location. Meurice et al. [14] then extended this work, using a more powerful analysis to determine the tables and columns accessed by queries. Similarly to WAFA, van den Brink et al. [15] identified query strings by matching SQL keywords; they then used a backward analysis to find variables holding these queries and a forward analysis to find all pieces of the query. The focus of their work was on measuring the quality of the data access code in a program. Ngo and Tan [16] used symbolic execution and a reduced control flow graph, called an Interaction Flow Graph (IFG), to extract information on queries built along different paths through the program. Due to the possibly exponential nature of the analysis, they used heuristics to properly handle function calls, and the analysis was terminated in cases where it ran for too long, leading to possible false positives. Gould et al. [17] used a string analysis to build a finite state automaton representing the possible queries at each *hotspot* where a query could be invoked. This is then processed using a context-free reachability algorithm to determine possible type errors, based on the actual database schema used by the program. Our work differs from the above work in that our main goal is to classify queries into patterns based on how they are built, which will then drive future decisions about the specific types of analysis needed for program comprehension and refactoring.

Looking more broadly, there have been a number of studies that use static, or a mix of static and dynamic, techniques to

examine how language features are used or to find patterns that can be exploited in analysis. Static approaches include the work of Hackett and Aiken on aliasing patterns for C systems programs [18]; the work of Ernst et al. [19] and Liebig et al. [20] on usage of the C preprocessor; the work of Collberg et al. on Java bytecode [21]; and the work of Baxter et al. on characterizing the distributions of metrics computed over Java bytecode and Java source code [22]. Dynamic or mixed techniques include the work of Knuth on FORTRAN [23]; the work of Richards et al. on the use of dynamic features in JavaScript [24]; the work of Morandat et al. on R [25]; and the work of Furr et al. on Ruby [26].

The work in this paper is built on our earlier work on feature usage in PHP systems [1], [2] and program analysis for PHP [27], [28], and makes use of PHP AiR [5], [6], our framework for PHP program analysis in Rascal. Rucareanu, as part of her work on refactoring MySQL queries in PHP [3], used PHP AiR to identify several query patterns and explored how they could be refactored. The work here has focused on expanding this collection of patterns and exploring their occurrences in actual PHP code in more detail.

VII. Conclusions and Future Work

Although we believe these initial results already give some insight into how queries are built in PHP code, there are opportunities to further expand this research. First, we would like to expand the number of systems studied, both to confirm the patterns that we have already found and to potentially find new patterns. Second, we would like to more precisely classify how dynamic fragments are used in forming SQL queries, something that will be critical for better understanding the queries and for transforming them to use new libraries. Third, we would like to strengthen the analysis used in pattern detection to identify possible indirect changes to query variables, such as through aliasing, since this will be needed for situations where these patterns are driving code transformations.

Our longer term goal is to take advantage of these patterns in building effective source transformation tools for PHP, specifically for evolving existing systems to use newer, more modern database APIs. Better understanding these patterns will allow us to more precisely handle scenarios that occur in practice, allowing us to focus our efforts. We also believe this will be helpful in building code comprehension tools, which can inform developers of the links between queries and calls to database APIs and also provide models of these queries that can improve developer understanding of existing code.

Acknowledgments

We would like to thank Ioana Rucareanu, whose earlier work on identifying MySQL query patterns inspired the ongoing work described in this paper. This research is supported in part by NSF grants CCF-1262933 and CCF-1560037.

References

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

[2] M. Hills, "Evolution of Dynamic Feature Usage in PHP," in *Proceedings of SANER 2015*. IEEE, 2015, pp. 525–529.

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

[4] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic Creation of SQL Injection and Cross-Site Scripting Attacks," in *Proceedings of ICSE 2009*. IEEE, 2009, pp. 199–209.

[5] M. Hills and P. Klint, "PHP AiR: Analyzing PHP Systems with Rascal," in *Proceedings of CSMR-WCRE 2014*. IEEE, 2014, pp. 454–457.

[6] M. Hills, P. Klint, and J. J. Vinu, "Enabling PHP Software Engineering Research in Rascal," *Science of Computer Programming*, vol. 134, pp. 37–46, 2017.

[7] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM 2009*. IEEE, 2009, pp. 168–177.

[8] A. Cleve and J. Hainaut, "Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering," in *Proceedings of WCRE 2008*. IEEE Computer Society, 2008, pp. 192–196.

[9] N. Noughi, M. Mori, L. Meurice, and A. Cleve, "Understanding the Database Manipulation Behavior of Programs," in *Proceedings of ICPC 2014*. ACM, 2014, pp. 64–67.

[10] L. Meurice, "Visualizing SQL execution traces for program comprehension," Master's thesis, University of Namur, 2013.

[11] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "WAFA: Fine-grained Dynamic Analysis of Web Applications," in *Proceedings of WSE 2009*. IEEE Computer Society, 2009, pp. 141–150.

[12] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[13] C. Nagy, L. Meurice, and A. Cleve, "Where Was This SQL Query Executed? A Static Concept Location Approach," in *Proceedings of SANER 2015*. IEEE Computer Society, 2015, pp. 580–584.

[14] L. Meurice, C. Nagy, and A. Cleve, "Static Analysis of Dynamic Database Usage in Java Systems," in *Proceedings of CAiSE 2016*, ser. LNCS, vol. 9694. Springer, 2016, pp. 491–506.

[15] H. van den Brink, R. van der Leek, and J. Visser, "Quality Assessment for Embedded SQL," in *Proceedings of SCAM 2007*. IEEE Computer Society, 2007, pp. 163–170.

[16] M. N. Ngo and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Information & Software Technology*, vol. 50, no. 3, pp. 160–175, 2008.

[17] C. Gould, Z. Su, and P. T. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," in *Proceedings of ICSE 2004*. IEEE Computer Society, 2004, pp. 645–654.

[18] B. Hackett and A. Aiken, "How is Aliasing Used in Systems Software?" in *Proceedings of FSE 2006*. ACM, 2006, pp. 69–80.

[19] M. D. Ernst, G. J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[20] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines," in *Proceedings of ICSE 2010*. ACM, 2010, pp. 105–114.

[21] C. S. Collberg, G. Myles, and M. Stepp, "An empirical study of Java bytecode programs," *Software: Practice and Experience*, vol. 37, no. 6, pp. 581–641, 2007.

[22] G. Baxter, M. R. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. D. Tempero, "Understanding the Shape of Java Software," in *Proceedings of OOPSLA 2006*. ACM, 2006, pp. 397–412.

[23] D. E. Knuth, "An Empirical Study of FORTRAN Programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *Proceedings of PLDI 2010*. ACM, 2010, pp. 1–12.

[25] F. Morandat, B. Hill, L. Osvald, and J. Vitek, "Evaluating the Design of the R Language - Objects and Functions for Data Analysis," in *Proceedings of ECOOP 2012*, ser. LNCS, vol. 7313. Springer, 2012, pp. 104–131.

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

[27] M. Hills, "Variable Feature Usage Patterns in PHP," in *Proceedings of ASE 2015*. IEEE, 2015, pp. 563–573.

[28] M. Hills, P. Klint, and J. J. Vinju, "Static, Lightweight Includes Resolution for PHP," in *Proceedings of ASE 2014*. ACM, 2014, pp. 503–514.