Supporting Analysis of SQL Queries in PHP AiR

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

***Abstract*—The code behind dynamic webpages often includes calls to database libraries, with queries formed using a combination of static text and values computed at runtime. In this paper, we describe our work on a program analysis for extracting models of database queries that can compactly represent all queries that could be used in a specific database library call. We also describe our work on parsing partial queries, with holes representing parts of the query that are computed dynamically. Implemented in Rascal as part of the PHP AiR framework, the goal of this work is to enable empirical research on database usage in PHP scripts, to support developer tools for understanding existing queries, and to support program transformation tools to evolve existing systems and to improve the security of existing code.**

I. Introduction

PHP, one of the most popular languages for open-source development,[1] is commonly used to build dynamic websites, where the information displayed on the page is based on user inputs and data stored in a (usually) relational database. Access to this data is through a variety of database libraries, either provided with the PHP language or available as separate downloads. Commonly-used libraries include the original MySQL library,[2] the MySQL Improved (MySQLi) library,[3] and the PHP Data Objects (PDO) library.[4] Although these libraries have different APIs, all perform queries in a similar way: a query string is built using a combination of string literals—representing static parts of the query—and PHP expressions—representing dynamic parts of the query, such as the actual values used in a `WHERE` clause of a `SELECT` statement.

As part of our current research, we are exploring how developers create queries that use the original MySQL library [1]. This library, available since version 2 of the PHP language, provides a procedural interface for querying MySQL databases. Queries are executed with the `mysql_query` function by passing a query string to the function, and features such as prepared statements are not supported. Even though the original MySQL library is now deprecated in PHP 5 releases from 5.5.0 and is no longer supported in PHP 7, a significant body of code still uses this library, including commonly-used systems such as the WordPress blogging platform. The ultimate goals of this research are to help developers better understand queries they find in code, to enable empirical research on how database queries are used in PHP code, and to renovate existing systems to allow the use of more modern and more secure database libraries such as MySQLi and PDO.

[1]https://octoverse.github.com/
[2]http://php.net/manual/en/book.mysql.php
[3]http://php.net/manual/en/book.mysqli.php
[4]http://php.net/manual/en/book.pdo.php

Our approach, described below, is to statically extract a model representing the queries that can be passed to a specific occurrence of a query function such as `mysql_query`. This model can be used to better understand how the query is built in the code, and can also be used to statically generate the variants of the query, built based on different program paths taken at runtime (e.g., a query may have different `WHERE` clauses, based on different branches of a conditional). These variants can then be parsed using an extended MySQL parser that works with incomplete queries, where dynamic parts of the query are replaced with "holes". The resulting MySQL ASTs can then be used to get more accurate information about which parts of the query are given statically and which parts (e.g., parameters, column names) are provided at runtime.

The rest of the paper is organized as follows. In Section II we provide an overview of the process used to extract query models and to parse the partial queries yielded by a model. Section III and Section IV then provide detailed explanations: Section III explains how query models are extracted from PHP code and how these models are turned into a set of possible queries, while Section IV explains how queries with holes representing dynamically-provided information are parsed and represented as MySQL ASTs. We then briefly discuss related work in Section V, while Section VI discusses possible future work and concludes.

The work described below is mainly implemented in the Rascal meta-programming language [2], [3] using the PHP Analysis in Rascal (PHP AiR) framework [4], [5]. The analysis and parsing code described in this paper is available online.[5]

II. Overview

Figure 1 provides an overview of the process used to build a model and extract and parse the possible queries used in a database library call at a specific program point. The first step in the process is to parse the PHP scripts that make up the system being analyzed. This is done using our fork[6] of an open-source PHP parser[7], written in PHP. This generates a value of Rascal type `System`, which represents the parsed system and includes ASTs for each of the parsed PHP scripts.

Using these ASTs, and the location of a call to a database query API function such as `mysql_query` (part of the original MySQL API), the Model Builder statically extracts a model representing the actual SQL queries that could be passed to this function, using a process described in Section III. This

[5]https://github.com/ecu-pase-lab/mysql-query-construction-analysis
[6]https://github.com/cwi-swat/PHP-Parser
[7]https://github.com/nikic/PHP-Parser/

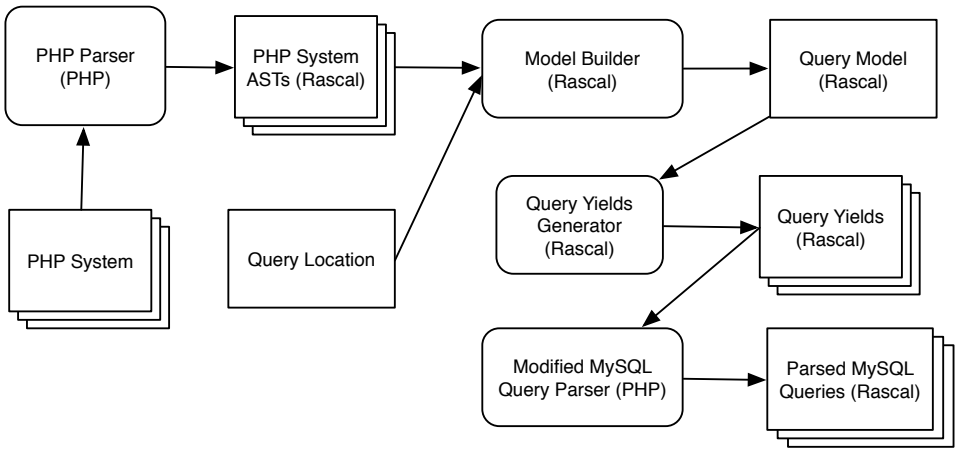

Fig. 1. Overview: Extracting Query Models and Parsing Modeled Queries.

model differentiates between static and dynamic parts of the queries, provides links from names used in the query to the values that these names represent, and includes information about the conditions under which certain parts of a query are present. To get each of the possible queries represented by the model, the Query Yields Generator builds a representation—here called a *yield*—of each query, with each yield reflecting a specific query executed based on a specific program path from the start of the surrounding context (e.g., the function containing the API call) to the API call. As with the model, each yield includes a combination of static and dynamic pieces.

To parse each of these queries, we use our fork[8] of the open-source MySQL parser[9] released as part of phpMyAdmin,[10] a web-based tool for managing MySQL databases. This parser has been modified to properly parse queries with "holes" representing the dynamic parts of the query. The result of running the parser on each yield is to generate a Rascal AST for each query. At this time only queries using the MySQL dialect of SQL are supported, but similar parsers could be used for other dialects. More details on how the yields of the model are parsed, and on how each parsed query is represented in PHP and in Rascal, are presented in Section IV.

Our current corpus for testing the model extraction and parsing processes uses a total of 17 systems with 749,451 lines of PHP code and 4,788 PHP files, as counted by the cloc[11] tool. In total, these systems contain 2,607 calls to `mysql_query`. We plan to extend this to additional systems in the future.

III. Query Models

As was described in Section II, a query model is used to statically represent the different queries that could be passed to a database query function such as `mysql_query` (hereafter referred to just as the *query call*). These differences reflect different control flow paths that reach the query call. In this section, we describe how the query model is built and how this is used to generate the possible individual queries (called *yields*) that could actually be executed.

```
literalFragment(str literalFragment)
compositeFragment(list[QueryFragment] fragments)
concatFragment(QueryFragment l, QueryFragment r)
nameFragment(Name name)
dynamicFragment(Expr fragmentExpr)
```

Fig. 2. Query Fragment Constructors in PHP AiR.

A. Representing Queries

A query call in a PHP script accepts an expression that evaluates to a string containing the query text. This expression can be a literal string, but is often more complex, consisting of both static and dynamic *fragments* and built across multiple assignments that may involve control flow. In PHP AiR, each expression that contributes to the query is modeled as a `QueryFragment`. The definition of these fragments is shown in Figure 2. A query fragment can represent a string literal (`literalFragment`), string interpolation (`compositeFragment`, with child fragments for each embedded expression), string concatenation (`concatFragment`, with fragments for the left and right operands), the use of a name (`nameFragment`), or the use of another expression not explicitly modeled (`dynamicFragment`, containing the dynamic expression). Before transforming a query expression into a `QueryFragment` a number of simplifications are applied to the expression, including simulating library functions that work over strings, replacing constants, and performing operations such as string concatenation over string literals.

B. Building Query Models

Algorithm 1 describes the process of building a query model to represent the queries that could be passed to a query call. The algorithm takes the PHP system containing the query call (sys) and the location of the query call ($callLoc$) as parameters. The

[8]https://github.com/ecu-pase-lab/sql-parser
[9]https://github.com/phpmyadmin/sql-parser
[10]https://www.phpmyadmin.net/
[11]https://github.com/AlDanial/cloc

Input : *sys*, a PHP system, mapping from file locations to abstract syntax trees
Input : *callLoc*, a location indicating the query call to be analyzed
Output : *res*, a query model

```
1  inputCFG ← buildCFG4Loc (callLoc)
2  inputNode ← the CFG node from inputCFG representing the call at callLoc
3  d ← definitions (inputCFG)
4  u ← uses (inputCFG, d)
5  slicedCFG ← basicSlice (inputCFG, inputNode, usedNames (u, inputNode), d, u)
6  startingFragment ← expr2qf (inputNode)
7  fragmentRel ← {}
8  while fragmentRel continues to change do
9  |  nodesToExpand ← (inputNode.l × startingFragment) ∪ fragmentRel⟨3, 4⟩
10 |  foreach (nodeLabel × nodeFragment) ∈ nodesToExpand do
11 |  |  add expandFragment (nodeLabel, nodeFragment, d, u) to fragmentRel
12 |  end
13 end
14 fragmentRel ← addEdgeInfo (fragmentRel, slicedCFG)
15 res ← the model, including fragmentRel, startingFragment, and callLoc
```

Algorithm 1: Extracting a Model of a SQL Query.

actual implementation of the algorithm also takes a collection of cached analysis results (e.g., already extracted control flow graphs) to improve performance, but this is just a performance improvement and is not shown here.

Given these two inputs, the algorithm starts by building an intraprocedural control-flow graph (CFG) for the script, function, or method containing the query call (Line 1). The CFG node representing the query call is then assigned into *inputNode*, since this is needed later in the analysis (Line 2). Using the CFG, a def-use analysis then computes the definitions (Line 3) and uses of these definitions (Line 4) for all nodes in the graph. A basic slicing algorithm is then used to remove extraneous nodes from the CFG (Line 5). This algorithm is given the CFG, the starting node, the names used in this node (i.e., in the query expression), and the def-use information, and then computes a backwards, intraprocedural slice, starting at the query call. Any node not included in the slice is discarded, leaving only statements and expressions that actually impact the expression used in the query call.

With this information, the fragment relation *fragmentRel* can be computed. *fragmentRel* is defined as a relation of type $(Lab \times QueryFragment \times Name \times Lab \times QueryFragment \times EdgeInfo)$, meaning that it maps a *Name* (projection 2) used at a given CFG node (represented by the unique node label of type *Lab*, projection 0) and query fragment (projection 1) to another CFG node (projection 3) and query fragment (projection 4) that provides a definition for this name. Viewing this as a graph, each label and query fragment combination can be seen as a node, with directed edges from a name use to a name definition, and with additional labels (*EdgeInfo*) possibly present on each edge.

The first step is to compute *startingFragment*, the `QueryFragment` representing the query expression used in the query call (Line 6). To link names in this fragment to their definitions, *fragmentRel* is computed in a fixpoint operation: starting with a tuple containing the label of the CFG node representing the query call (accessed using field `l`) and the fragment for this call (Line 9), `expandFragment` will first return mappings from names using in the fragment to the fragments that define these names (based on def-use information, Line 11). Names used in these fragments will then be expanded to include the fragments that define them (Lines 10 through 12, which expand one fragment at a time—the CFG node represented by a fragment, and the fragment to expand, are in projections 3 and 4 of *fragmentRel*, respectively). *fragmentRel* grows monotonically until no new mappings are added, meaning that all names are linked to the nodes that define them, ending the loop started on Line 8. Names that ultimately link back to formal parameter or global declarations will expand to `globalFragment`s and `inputParamFragment`s, respectively, while names with no definitions are represented as `unknownFragment`s.

Once the *fragmentRel* has been computed, `addEdgeInfo` uses the sliced CFG to add predicates to each edge representing the conditions that must be true for those edges to be reached (Line 14). The query model, including this decorated version of the relation, the location of the query call, and the fragment representing the initial query expression, are then returned as the result of building the model (Line 15).

C. Transforming Models to Queries

The query model represents all possible queries that can be passed to the query call, also referred to here as the *yields* of the model. Each yield is defined as a list of static and dynamic pieces, with different yields corresponding to different runtime execution paths. The yields are computed by starting at the fragment representing the initial query expression and then following all possible paths from this fragment through the model graph. Paths that would lead to cycles are based on names defined (directly or indirectly) in terms of

themselves, e.g., variables that are built up through multiple string concatenations inside a loop. When this occurs the path is cut to break the cycle, leaving such names unexpanded. While it is possible to allow a limited expansion in such cases, this is not currently implemented. Condition information, marked on the edges, is taken into account to remove infeasible yields, such as those using assignments from both branches of a conditional in a single yield.

IV. Parsing SQL Queries

As mentioned in Section III, the set of yields for a query model represents all possible queries executed by the query call. Based on our prior results [1], a large majority of the queries used in actual systems are formed using a combination of both static SQL text, given as string literals, and expressions that dynamically provide parts of the query at runtime. This leads to partial queries, with "holes" representing these dynamic pieces of information. To properly support studies of how SQL is used in PHP code and to create accurate transformation tools, we have developed support for parsing these partial queries, with the parsed form given as an AST in PHP AiR. The rest of this section describes how this is accomplished.

A. Converting Yields to Strings

To generate string forms of yields that can be parsed, we defined the following conversion. For static pieces, the literal query fragment is used. Name pieces and dynamic pieces are replaced with a `?` followed by an integer. Figure 3 (from system WebChess 0.9.0 in our corpus) provides an example of this conversion. The top of the figure shows the yield, which includes five pieces: three static pieces, representing the static query text used in the query (here, to insert values into a table), and two dynamic pieces, here standing for two variables used in the code to provide the values being inserted. The string representing this query is then given at the bottom of Figure 3. The three static pieces of the yield remain unchanged while the two name pieces are replaced by numbered query hole symbols, `?0` for `_SESSION` and `?1` for `CFG_MINAUTORELOAD`.

```
[
  staticPiece("INSERT INTO preferences (playerID,
      preference, value) VALUES ("),
  namePiece("_SESSION"),
  staticPiece(", \'autoreload\', "),
  namePiece("CFG_MINAUTORELOAD"),
  staticPiece(")")
]

INSERT INTO preferences (playerID, preference, value)
    VALUES (?0, 'autoreload', ?1)
```

Fig. 3. Converting SQL Yields to Strings.

B. The Parser

As mentioned in Section II, to parse partial queries we are using a custom fork of the MySQL parser included with phpMyAdmin. This parser focuses specifically on the MySQL dialect and has been used by phpMyAdmin since version 4.5. Since this parser is designed for static queries, modifications were necessary to adapt it to work with queries with dynamic holes. These modifications generally fell under one of two categories:

1) Modifying the grammar and parsing logic to work with query hole symbols
2) Writing functions to convert parsed queries to Rascal terms

Productions in the parser are not specified using a BNF-like syntax, but are instead encoded into parse methods in classes representing the non-terminal type being parsed. For instance, a `SELECT` query is represented by a class named `SelectStatement`, while an expression in such a statement is represented by a class named `Expression`. The first category of modifications began with the addition of a new query hole terminal (e.g., `?1`, `?2`) recognized during lexical analysis. Modifications were then made to multiple classes to allow query holes to be treated as valid expressions. This allows the parser to be able to recognize a query hole wherever expressions are expected, such as in the following generic select statement `SELECT ?0 FROM...`. The hole `?0` in this case provides the value of the column or columns to be selected.

Another major modification to the parser was creating new classes for SQL conditions that most commonly occur in SQL `WHERE` clauses. Our current results [1] show that a large number of query holes are contained in SQL conditions. These classes support analysis of query hole placement in SQL conditions more intuitively than the original data structure, which was based around a complex collection of nested arrays. At this point, the parser supports the following condition types:

- `Expr [NOT] BETWEEN Expr AND Expr`
- `Expr IS [NOT] NULL`
- `Expr [NOT] IN (Expr [, Expr] ...)`
- `Expr [NOT] LIKE Pattern`
- Simple comparisons such as `Expr1 = Expr2` and `Expr1 < Expr2`

We focused on these condition types since they are the most common in the systems we have analyzed. Support for additional condition types is part of our current work. Figure 4 includes an example of the internal PHP AST—including our modifications—used by the parser for the compound condition `WHERE floors BETWEEN 1 and ?1 AND roof = ?0`. The namespace prefixes have been removed from the class names to make them easier to read.

The second category of modifications involved adding support for outputting parsed queries as Rascal terms representing MySQL ASTs. This was done by writing functions in the parser that convert the PHP objects output by the parser to a string representation of an equivalent Rascal term representing the AST—the same process as is used to generate Rascal ASTs for parsed PHP scripts. An example of such a conversion can be found in Figure 5, which represents the resulting AST for the SQL condition found in Figure 4. Our abstract syntax defined in Rascal is discussed in more detail below.

```
object(ConditionNode)#43 (3) {
      ["value"]=> string(3) "AND"
      ["left"]=>
      object(ConditionNode)#41 (3) {
        ["value"]=>
        object(BetweenCondition)#42 (5) {
          ["not"]=> bool(false)
          ["expr"]=> string(6) "floors"
          ["lowerBounds"]=> string(1) "1"
          ["upperBounds"]=> string(2) "?1"
        }
        ["left"]=> NULL
        ["right"]=> NULL
      }
      ["right"]=>
      object(ConditionNode)#38 (3) {
        ["value"]=>
        object(ComparisonCondition)#39 (5) {
          ["not"]=> bool(false)
          ["lhs"]=> string(4) "roof"
          ["op"]=> string(1) "="
          ["rhs"]=> string(2) "?0"
        }
        ["left"]=> NULL
        ["right"]=> NULL
      } }
```

Fig. 4. Example PHP Object Representing a SQL Condition.

C. Representing MySQL ASTs in Rascal

The MySQL AST type in Rascal, `SQLQuery`, includes constructors for each query type (`SELECT`, `UPDATE`, etc). Each constructor contains fields for the clauses a particular query type can contain. Furthermore, clauses are also represented using constructors, with fields for each part of the clause. This is illustrated in Figure 6 (from system Schoolmate 1.5.4 in our corpus) which shows an `UPDATE` query (elided in the middle) with dynamic holes and its corresponding representation in Rascal. The Rascal representation contains values corresponding to the table name, `SET` operations, `WHERE` clause, and `LIMIT` clause found in the original query. Figure 6 also shows that the holes in the original query `SET` and `WHERE` clauses are accurately represented in the AST. This supports empirical research into how developers build queries by allowing quick analysis of which query parts are static and which come from dynamic sources. Checking if a field in a particular clause is a hole involves a simple check on the string representing the value.

Our current research shows that most query holes are either used as parameters (such as all the holes in Figure 6) or contain the value of an identifier (such as `SELECT ?0 FROM...`). However, we have found some infrequent cases where an entire

```
where(and(
 condition(between(false,"floors","1","?1")),
 condition(simpleComparison("roof","=","?0"))
))
```

Fig. 5. Example Rascal Term for a SQL Condition.

```
UPDATE schoolinfo SET schoolname = "?0",
 address = '?1' ... fpoint = '?11'
 where schoolname = '?12' LIMIT 1

updateQuery(
 [name(table("schoolinfo"))],
 [setOp("schoolname","?0"),
  setOp("address","?1") ...
  setOp("fpoint","?11")
 ],
 where(condition(
  simpleComparison("schoolname","=","?12"))),
 noOrderBy(),
 limit("1"))
```

Fig. 6. An Update Query with Corresponding Rascal AST.

clause or chunk of query text is contained in a query hole. Representing these cases is part of our current work.

V. Related Work

Our work is most closely related to that of Meurice et al. [6] and of Nagy et al. [7]. In the first, the authors describe a static analysis for Java that can recover complete query strings used with the JDBC, Hibernate (a popular ORM), and JPA (a Java standard for object persistence) libraries. It also provides details about schema objects used by the queries, something we do not currently do but that should be possible to extract from the ASTs we generate for each query. Unresolved dynamic query fragments are not currently modeled, leading to some cases where executed queries using these features are not recovered by the analysis. In the second, the authors have created a static analysis to identify the set of program locations where a specified query could be executed. For a given query call, they can generate a string representing the query, with a placeholder representing holes (referred to as "unresolved query fragments" in the paper) in the query. They also have a modified MySQL parser to parse these queries, with *Joker* nodes representing parts that cannot be parsed. The motivation for their work is to help developers identify where program concepts related to a specific query are implemented in the system. In our case we plan to use query models and ASTs of the queries for empirical research, to feed into program transformation tools, and to help developers understand how the queries used at a specific query call are built. We are also targeting a different language and different APIS— PHP and (at first) the PHP MySQL API, versus Java and the JDBC and Hibernate APIs.

Our work is also related to the work of Ngo and Tan [8]. In their work they statically analyze PHP code to extract information about all interactions with the database. This analysis uses symbolic execution over a reduced control flow graph, called an Interaction Flow Graph (IFG), that captures just the control flow nodes related to database interactions. Their analysis is a whole-program analysis, while ours is currently intraprocedural. This has been a conscious decision on our part—the file inclusion mechanism in PHP is dynamic and cannot always be resolved statically [9], meaning it isn't always possible to know the whole program in advance, or to know

in what contexts a specific file could be included. We are also exploring how often non-local information is used for query text versus for query parameters; the former is much more important for understanding and transforming queries. Similarly to our work on resolving file inclusion [9], we plan to develop an interprocedural version of the algorithm as well that will extend the existing models to account for values provided across call boundaries.

Finally, a number of approaches have focused on assessing code quality, finding errors in queries, or detecting potential security vulnerabilities. To measure the quality of database access code, van den Brink et al. [10] identified query strings by first matching SQL keywords in the program text; additional analysis was then applied to find related variables and other pieces of the query. Christensen et al. [11] developed a string analysis for Java, using this to detect errors in Java JDBC queries by modeling the generated strings and the allowed SQL syntax as regular approximations of context-free languages. A similar approach, which actually makes use of this string analysis, is used by Wassermann et al [12], [13] to detect type errors in query strings in Java programs. This string analysis is also used as part of Halfond and Orso's AMNESIA system [14] for preventing SQL injection attacks. In AMNESIA, an automaton is built to represent the possible queries at a given program point. This automaton is then used at runtime to ensure that actual queries have not modified the query structure, a pattern common to SQL injection attacks. Su and Wassermann [15] use a similar insight with their SQLCHECK algorithm, which checks queries to ensure they do not modify the query structure before allowing them to be executed.

The work in this paper makes use of PHP AiR [4], [5], our framework for PHP program analysis in Rascal. The query models and query parser described above will be used to extend our earlier work on identifying query construction patterns [1], which instead used a combination of pattern matching and regular expression matching to extract information about MySQL queries in PHP code.

VI. Future Work and Conclusions

In this paper we have presented a program analysis for building query models in PHP code. We have also described how these models are used to produce the queries represented by the model, and have explained how we have modified an existing MySQL parser to parse queries with missing information (described above as holes) and produce MySQL ASTs defined according to a collection of Rascal types.

One motivation for this work was to extend our earlier work on identifying query construction patterns in PHP code [1], mentioned above. Over 80% of the queries examined in this earlier work fell into a general "dynamic" pattern, meaning that the query was made up of unresolved dynamic pieces. Having a consistent representation for all queries should make identifying both new and existing patterns easier, and should allow us to make finer-grained distinctions in how dynamic queries are formed. The use of a parser for identifying which parts of a MySQL query are dynamic instead of using regular expressions should yield more accurate results and give us more insight into the queries themselves, allowing us to extract information about the schema elements and SQL operations included in the query. The extracted models and queries, as well as information on usage patterns, will be used to create tools to help developers understand existing query code and to transform their code to use newer, more secure database libraries. We believe the work presented here will be useful for others trying to build tools that reason about database calls.

In the future we plan to extend this work to support other dialects of SQL and other data access libraries for PHP (such as MySQLi and PDO). This will mainly require adding the capability to use different parsers, since the code that is used to build the models can work with multiple SQL dialects and query libraries. We also plan to add improved visualization capabilities for viewing models and linking them back to the underlying code. Finally, we plan to implement an interprocedural version of the model construction algorithm that will account for values provided across call boundaries.

Acknowledgments

This research is supported in part by NSF grants CCF-1262933 and CCF-1560037.

References

[1] D. Anderson and M. Hills, "Query Construction Patterns in PHP," in *Proceedings of SANER 2017*. IEEE, 2017, pp. 452–456.

[2] P. Klint, T. van der Storm, and J. Vinju, "EASY Meta-programming with Rascal," in *Post-Proceedings of GTTSE'09*, ser. LNCS. Springer, 2011, vol. 6491, pp. 222–289.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise Analysis of String Expressions," in *Proceedings of SAS 2003*, ser. LNCS, vol. 2694. Springer, 2003, pp. 1–18.

[12] G. Wassermann, C. Gould, Z. Su, and P. T. Devanbu, "Static Checking of Dynamically Generated Queries in Database Applications," *ACM TOSEM*, vol. 16, no. 4, 2007.

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

[14] W. G. J. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," in *Proceedings of ASE 2005*. ACM, 2005, pp. 174–183.

[15] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Proceedings of POPL 2006*. ACM, 2006, pp. 372–382.