Supporting Analysis of SQL Queries in PHP AiR

David Anderson and Mark Hills (@hillsma on Twitter)

17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2017), Engineering Track
September 17-18, 2017
Shanghai, China

http://www.rascal-mpl.org

Motivation

- We want to transform uses of older database APIs into equivalent uses of newer, safer APIs

- We want to aid developers in understanding how queries are built in their (maybe unfamiliar to them!) programs

- (For us and other researchers/tool builders) We want to better understand how PHP, database APIs, and query languages are used in existing code to help us build better tools for program analysis, comprehension, and transformation

Context: PHP and MySQL

- MySQL API uses query functions (mysql_query) to execute queries
- Queries given as strings, formed using string building operations
- Queries often have a mixture of static and dynamic pieces

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

Note: Real but horrible query, this has a major security vulnerability…

Research questions

- R1: How can we model how queries are built?

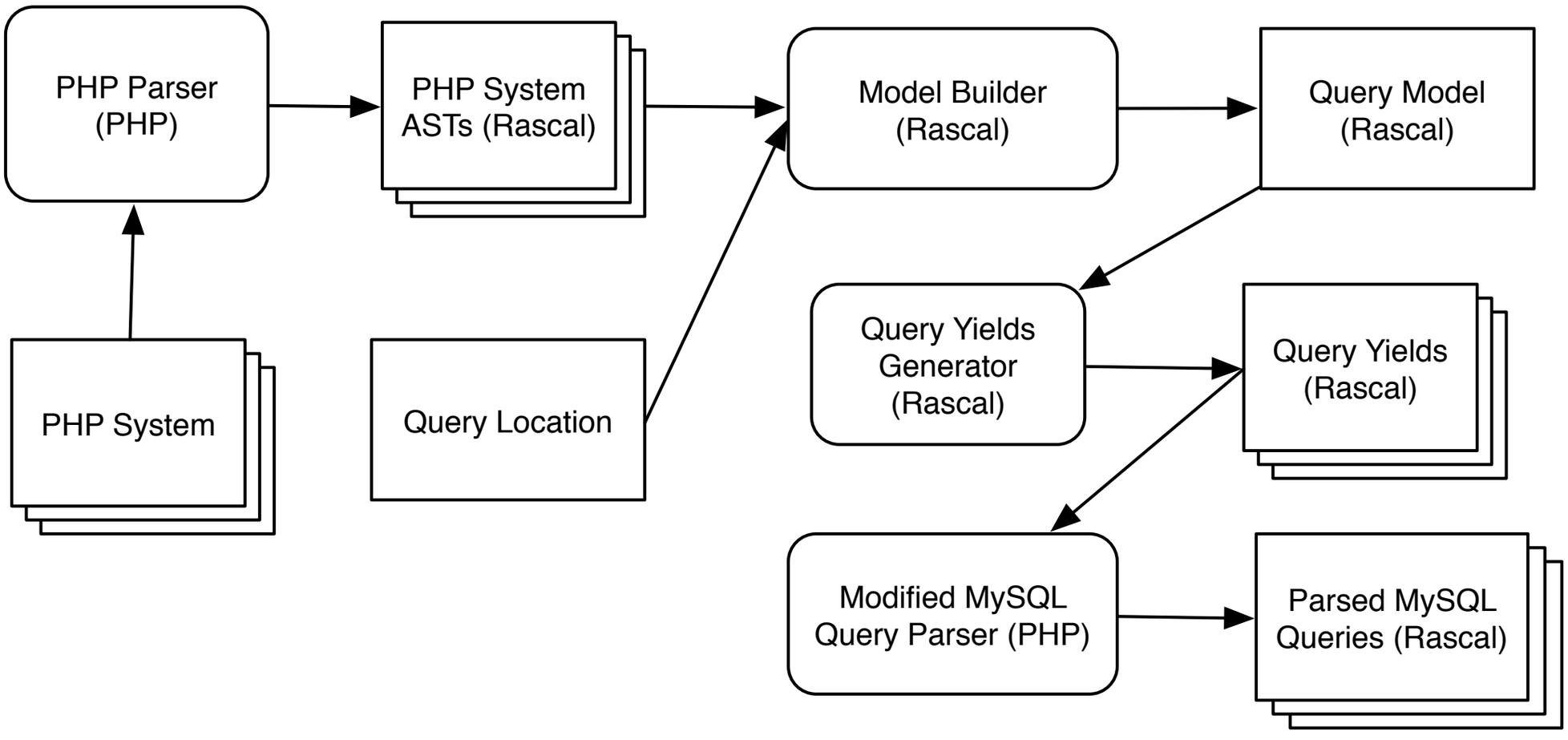
- R2: Using these models, how can we generate the queries or query templates (with placeholders for dynamic bits) that could actually be executed?

- R3: What parts of the query language (here, SQL) are used in these queries? Which of these parts are static, and which are dynamic?

Analyzing and parsing queries: methodology

- All analysis code is written using Rascal (http://www.rascal-mpl.org/), a meta-programming language for program analysis
  - https://github.com/ecu-pase-lab/mysql-query-construction-analysis
- PHP is parsed using a PHP parser written in PHP, parser yields Rascal terms
- MySQL queries are parsed using a fork of the parser found in phpMyAdmin, a web frontend for administering MySQL
- A quick note: this is all part of the PHP AiR project

Analyzing and parsing queries: “The Big Picture”



R1: Building models

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.

See the paper for all the details, here comes a summary…

R1: Building models

- Models are (possibly cyclic) graphs of *query fragments* (with a bit of bookkeeping info, like the location of the call)

- A query fragment is a static or dynamic piece of the query

- Intraprocedural, backwards slice throws away code that does not impact query

- CFG and def/use info link names in fragments to defs of those names

- Reachability conditions used to decorate graph edges

R2: Extract yields

- Yields are lists of “pieces”:
  - Static pieces for query text
  - Dynamic pieces for arbitrary expressions
    - Name pieces for names (useful to track separately)
- Generated by traversing the graph, currently cuts off when cycles detected
- Can use edge labels to filter infeasible yields, improving to work in more situations (loops are problematic)

R3: Parsing partial queries

- First, yields are converted to strings: static pieces yield strings directly, dynamic and name pieces are turned into query holes (e.g., ?1, ?2, generally ?n)

- Second, string is parsed by our modified MySQL parser, yielding PHP objects representing MySQL AST (limitation: we assume holes are expressions and do not cross clause boundaries, supporting this is ongoing work)

- Third, AST pretty-printed to a Rascal term representing AST, similarly to current PHP parser

Demo…

And now for some controversy

- We want to extend this to be interprocedural,
  but: for a really dynamic language, where even the decision of what code to include is deferred until runtime, is this even useful?

- To borrow from earlier: keep it simple! Do we even need to support the entire language for this to be useful for developers?

- For artifacts, are full VMs at all useful? Should we aim at using something like Docker? Images available in the cloud? Something else?

Discussion

Thank you!
Any Questions?

- Rascal: http://www.rascal-mpl.org
- Me: http://www.cs.ecu.edu/hillsma