Software Engineering — University of Amsterdam

Master Software Engineering

Master Thesis

PHP: Securing Against SQL Injection

:
Ioana Rucareanu

Supervisors: Dr. Jurgen Vinju and Dr. Mark Hills
Institute: Centrum voor Wiskunde en Informatica, Amsterdam

October 28, 2013

Contents

Abstract

PHP web applications have always been a preferred target of SQL injection attacks. Inadequate validation and sanitization of user inputs give rise to vulnerabilities even in widely-used websites of IBM, Hewlett-Packard, Cisco, WordPress and Joomla [1]. Various techniques based on static string analysis and taint-checking have been researched in order to detect the sensitive points of interaction with the database, that could injection favorize attacks. However, static solutions proved to produce false positives because of insufficient runtime informaion, whereas runtime techniques, although more precise, lead to a performance penalty and are disregarded because they require the addition of instrumented code.

We propose a method to fully remove the risk of SQL injection attacks, by relying on the use of prepared statements, which enforce SQL input to take only literal positions. Our prototype is able to refactor the mysql constructs from the PHP code and tranform them into parametrized queries supported by the PDO library. We based our implementation on the identification of certain patterns of query structures in the web applications, for which we provided and applied corresponding refactoring methods. Our static algorithm proved to transform correctly the set of programs we experimented with, but future work can extend its capabilities by relying on slicing techniques as well as offering interprocedural support.

Acknowledgement

Foremost, I would like to thank my thesis advisors Dr Jurgen Vinju and Dr Mark Hills for their continuous support and engagement through the research process of this thesis. I am highly grateful for the patience and availability they have shown even when communication was across multiple time zones. Without their feedback and reassurances, I would not have been able to finish this work in the short time I had reserved.

Besides my advisors, I would like to thank the entire SWAT team from CWI for introducing me into a cheerful academic working environment, and especially for their immediate availability whenever I requested their advice.

I am also grateful to Frank Tip, who offered several valueable suggestions for my research domain. Furthermore, I am indepted to all the members of the Master of Software Engineering at the University of Amsterdam for the education received and the opportunities offered in my study year.

My family deserves special thanks for their constant support and encouragements. To my friends, thank you for listening and keeping my spirits high during this whole process.

CHAPTER 1

Introduction

PHP is the most commonly preferred server-side programming language for generating dynamic content for web pages, being used by 80,5% of the websites world-wide [2]. Like with any other programming language, PHP code is as secure as the programmer writes it. PHP is preferred for its mild initial learning curve, built-in libraries for many common Web tasks, low hosting price, portability etc. A high usage rate by novices, not following best practice programming rules and missing out on the value of security awareness comes as a natural consequence. About 29% of all vulnerabilities listed on the National Vulnerability Database are linked to PHP [3], be it SQL injection (SQLi), Cross-Site Scripting (XSS), File disclosure or others.

PHP has natural integration with the SQL language, providing nearly native support for queries. Prior to the introduction of PHP 5.0, the interaction with the database system was achievable only through the low-level *mysql* API, by dynamically constructing query strings via string concatenation [4]. This also applies to general-purpose programming languages, such as Java [4], where JDBC is sometimes preferred over modern, more cumbersome APIs like JPA and Hibernate mainly because it requires no extra configuration and is easier to master. However, this low-level database interaction might lead to the problem of not properly sanitized inputs flowing into a query execution point (called *sensitive sink*). This can cause the web application to generate unintended output and is known as a *command injection attack* [4]. The main consequences for such type of attacks are: loss of confidentiality, stolen authentication, altering authorizations, affected database integrity [5].

At first glance, filtering the inputs and reject what it is *unsafe*, or altering the initial input values in order to make them *safe* (*sanitizing*) [6] might be seen as solutions to this problem. However, limiting the length of input strings, specifying patterns of what should be allowed/ avoided as it is considered to lead to SQLi attacks, escaping the user input by adding slashes, proved to be error-prone processes [7]. This is because a developer has to assess multiple attack scenarios in order to guarantee protection and then implement the strategies manually; proof that defensive coding is ineffective is provided in section 2.2.

Furthermore, a number of research studies have come up with both static and dynamic analysis tools in order to detect automatically the existing vulnerabilities in the code. However, none of the static approaches -using string analysis, taint checking- can fully guarantee the absence of SQLi vulnerabilities or avoid false-positives and this is mostly because some information is only available during program execution. Runtime techniques had better results, by checking actual runtime-generated queries, but they lead to a performance penalty [1]. This can be reduced when used in combination with static analysis methods [8] that could minimize the instrumented code's size.

1.1 Our solution

In this context, we identified the utility of building a prototype tool that would be able to refactor PHP code in order to completely remove the existing SQLi vulnerabilities. No security assumptions made, but a safe solution -transforming the queries into prepared statements. The difficulties of taint analysis, the task of performing correct escaping and type validation, the tedious specification of regular expression patterns that the unsafe input needs to conform to, they all can be avoided by applying this strategy.

Prepared statements were made available in PHP with the 5.0 version, via the *mysqli* and *PDO* libraries. They represent a database concept, based on precompiling: when you execute a SQL query, the database server will first prepare and cache an execution plan before executing the actual query. The SQL structure is therefore set in place and the input which is bound must fall into the category of query literals. In the world of software, the use of prepared statements is sometimes discarded because of this enforcement and also because of the extra time that precompiling requires. Details will be provided in section 2.3 of our report.

We implemented our prototype using the meta-programming language Rascal [9], specialized in code analysis and transformation. We replace the *mysql_* deprecated functions [10] with the new object-oriented functionality of PDO. We rely on the library's prepared statement support and dismiss the use of string concatenation and interpolation operations to bind query input. We promote this practice in order to remove the SQLi vulnerabilities from code. A syntactic query validation step has been included in this process, for reporting broken query structures.

The prototype had very good refactoring capabilities for the 4 projects we analysed. Where static information proves to be insufficient because of the different control flow paths the program could take at the execution time, an instrumentation-based solution is discussed. The query would then be refactored and validated at runtime, via prior statically placed instrumentation code. Moreover, we suggest an extension of the "only literals" constraint of prepared statements with table and column names support.

Our solution performs with a series of limitations. The algorithm remains intraprocedural and it assumes that all SQL structure is contained within string objects, built via assignment, concatenation or interpolation operations. Moreover, we assume that the variables incorporated into the query string expressions are placeholders for individual SQL input literals. As we mentioned, extension possibilities are discussed. Furthermore, the continous improvement of Rascal's PHP analysis capabilities could offer support for dealing with the unique challenges a language as dynamic as PHP presents: weak typing model ("duck typing"), operations that change their semantics with their operators' type, dynamic includes, variable variables etc.

1.2 Paper outline

The rest of the paper is structured as follows. We continue by presenting background information about SQL injection and study prevention methods. Chapter 3 discusses the related work done for exposing existing vulnerabilities in the code, followed by a refactoring solution that introduces prepared statements, via instrumented code. We present our algorithm in chapter 4, first describing the PDO library and the Rascal programming language. The Evaluation chapter 5 discusses our experimental methods and results. A discussion follows in the next section about possible improvement directions and we present our conclusions in chapter 7.

CHAPTER 2

SQL injection (SQLi)

An SQLi vulnerability results from allowing the data to enter an application from an untrusted source and using it to dynamically construct a SQL statement which is then executed. This permits the attacker to read sensitive stored data, alter the database information via INSERT/ UPDATE/ DELETE commands, corrupting its integrity, execute administration operations on the database, such as shutdown the DBMS and even more [5].

The simplest injection attack occurs in the absence of any checks. Consider this fragment of PHP:

```
mysql_query("select username, password from Student where id='".$_POST["id"].'"');
```

In this example, an attacker can provide as value for the request variable *id* a tautology in the form of **x' OR '1'='1** and retrieve all the existent usernames and passwords. By making use of the string delimiter ', the attacker modifies the intended syntactic structure of the query.

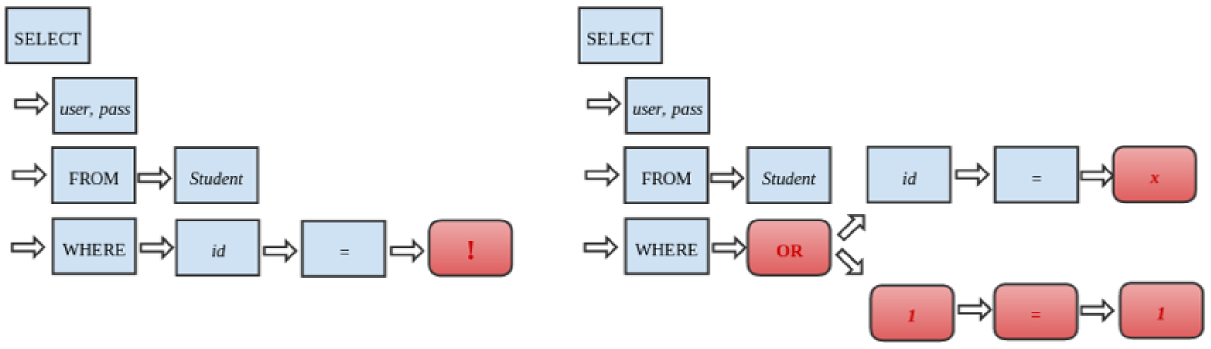

Figure 2.1: SQL Tree comparison after injection

Moreover, an attacker could even try to delete all the registrations by providing this value: **';DROP TABLE Student;–'**. The – characters indicate that the next parts of the query should be ignored. Fortunately, *mysql_query* does not allow the execution of multiple queries, but its new extension *mysqli* does that through the *mysqli_multi_query* function.

In the following sections we are going to present sanitisation techniques and insufficient examples of escaping. We will discuss reports showing the error-prone character of manual coding techniques and move forward to parametrized queries and the way we chose to address the problem of SQLi in our approach.

2.1 Input sanitisation

Sanitisation refers to cleaning user input to make it safe to use by the application. A key part in this process is **escaping** the values, for which PHP provides:

- general functionality like *addslahes()* and *magic_quotes()*, but their use has lead to some confusion because when used in combination, the slashes get duplicated [6]
- the mysql-specialised *mysql_escape_string()* and *mysql_real_escape_string()* functions, the latter providing escaping according to the current character set, thus being reccomended; what they do is prepending backslashes to the unsafe characters [7]
- the new *pdo::quote* function, which does not only escape the string, but it also quotes it [11]

However, escaping by itself can be insufficient for securing against SQLi. It has to be backed up by prior type validations and extra attention has to be paid to the correct use of string delimiters. Here are some examples of possible vulnerabilities, despite the use of *mysql_real_escape_string()* [1]:

- The function does not escape % and _ characters, and when these wildcards are used by an attacker and combined in a query structure with LIKE, GRANT or REVOKE, the sanitisation has no effect and the set of results can be more extended than initially intended.
- If not accompanied by type validation, it might prove ineffective. Consider the following piece of code:

  ```
  $id = mysql_real_escape_string($_GET["id"]);
  $query = "SELECT username, password FROM Student WHERE id = $id";
  ```

  And the value **1 OR 1=1** provided for the *id* variable. The escaping will have no value by itself and the tautology will again have the desired effect of retrieving all the passwords from the database. What is missing here is a type check for the input to be validated as a number, before applying the escape function. The problem could be detected if strings were not be seen as isolated lexical entities, but analysed with regard to the statement they generate together with the constant query structure [6].
- Absence or misuse of delimiters in query strings: the absence or misuse of delimiters could allow an injection attack and prove escaping and type checking useless [1].
  For the same code as above, the attacker might provide the encoded HEX string **0x270x780x270x200x4f0x520x200x310x3d0x31**. What would this mean? When the database server has the automatic type conversion function enabled, by using an alternate encoding method, the attacker would circumvent input sanitization routines. The above string would be converted by the database parser into the *varchar* value, that is the tautology string **'x' OR 1=1**. Because the conversion occurs in the database, the server program's escaping function would not detect any special characters encoded in the HEX string [1].

2.2 How effective is manual sanitisation?

It is clear that a correct sanitisation requires a high level of security awareness in PHP and enough time allocated in the process of software development. Fonseca and Vieira [7] provide a series of results that show how error-prone applying manual defensive pratices can be, as observed in 2008 in six well-known web-applications: PHP-Nuke, Drupal, PHP-Fusion, WordPress, phpMyAdmin and phpBB.

When security vulnerabilities are discovered in systems, version updates or patches are built for correcting them. Because the web applications analyzed by Fonseca and Vieira [7] are open-source and highly-used, a number of 655 patches had been disclosed to the public by 2008 and became the object of research. The number of vulnerabilities per projects: PHP-Nuke, Drupal, P|HP-Fusion, Wordpress, phpMyAdmin and phpBB, with the specification of the versions affected can be found in Table 8.1, in Appendix.

The 655 XSS and SQL injection security fixes were classified into 12 fault types, whose explanation (Table 2.1) and distribution (Figure 2.2) can be found below. Not surprisingly, 70,53% of the vulnerabilities favorized XSS attacks, mainly because with XSS, any input variable can become an attack entry point, in contrast with SQLi, which only targets unsafe queries [7].

Fault Type	Description
MFC	Missing function call
MFC extended	Missing function call extended
MVIV	Missing variable initialization using a value
MIA	Missing if construct around statements
MIFS	Missing if construct plus statements
MLAC	Missing "AND EXPR" in expression used as branch condition
MLOC	Missing "OR EXPR" in expression used as branch condition
WVAV	Wrong value assigned to variable
WPFV	Wrong variable used in parameter of function call
WFCS	Wrong function called with same parameters
ELOC	Extraneous "OR EXPR" in expression used as branch condition
EFC	Extraneous function call

Table 2.1: The fault types observed in the field, with their description [7]

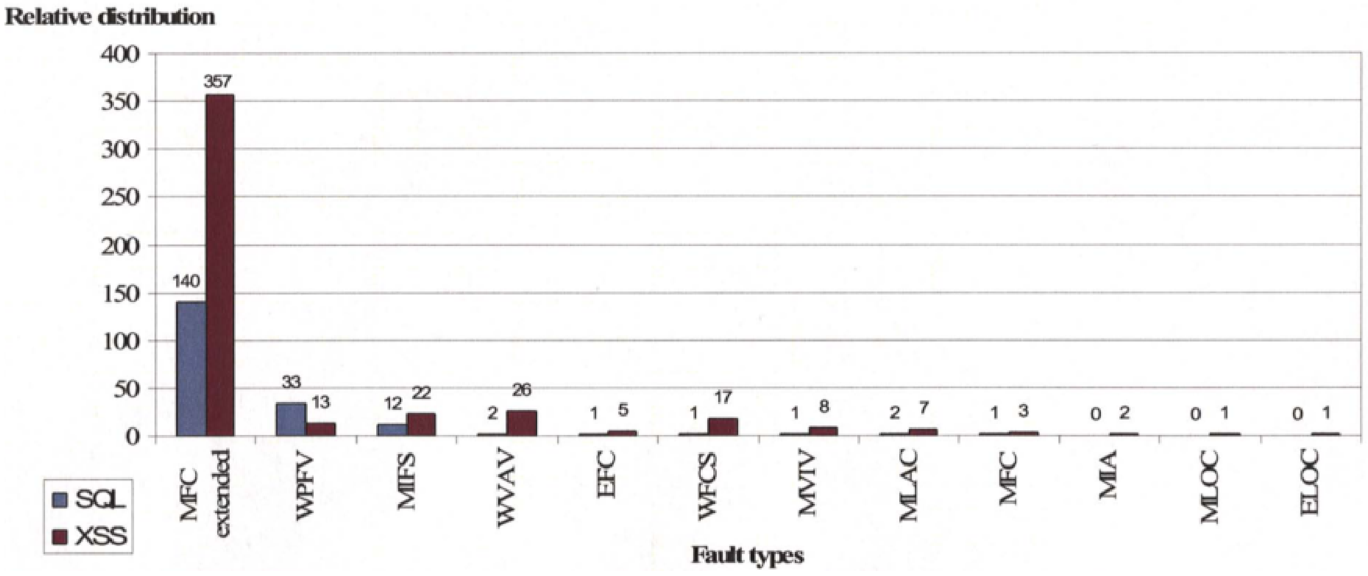

Figure 2.2: Vulnerability fault types summary [7]

The results showed that MFC extended with 75,88% is the most widespread fault type. MFC extended is based on MFC fault. It affects the validation and the cleaning process of both user inputs and database records, files, etc., thus leading to XSS or SQLi attacks. In many cases, also type checking is done via external functions, therefore omitting to include the required function calls will favorize string injection [7]. According to Fonseca and Vieira [7], WPFV, MIFS and WVAV are the next most encountered fault types, commonly caused by:

1. An omitted *'* around a PHP variable in SQL queries, allowing an attacker to inject a custom query.
2. An omitted *if* around a statement, that would perform a required null check, before applying sanitization.
3. A poor regular expression (regex) string used to filter the user input.

When the results were compared with other studies of common software faults, Fonseca and Vieira found considerable differences in their distributions. One research even introduced a new fault type with 16.10% coverage from the total number of faults- WLEC (Wrong logical expression used as branch condition). This shows that not all the possible fault types that can generate vulnerabilities are actually responsible for the security problem of web applications found in the real world [7].

It is important to understand that every existing vulnerability in the software systems analysed by these studies weakened the security of thousands of websites developed using a given version of the web application. Moreover, there are cases when web site administrators do not even update their sites' software when new patches and releases become available, maintaining the security flaws [7].

The reports presented in this section were meant to prove that relying on manual defensive programming techniques as validation and escaping cannot guarantee the application's safety, as developers can make omissions in their code. This consitutes a strong motivation to lean on the prepared statement's support.

2.3 The prepared statements solution

Prepared statementes, also known as *parametrized queries*, are intended to separate the SQL constant structure by the actual SQL input. At their creation time, the statements are *prepared* by statically specifying their constant SQL part and including placeholders to indicate where input should go. This is followed by binding statements, that map input variables to every placeholder now in the query string. When executing a statement for the first time, it is sent to the database, which compiles, optimizes its plan and saves the structure for future calls. Only afterwards the statement is run with the specified input variables. However, the next executions of the same statement, called with different parameters will skip the precompiling phase and improve the speed [12].

This mechanism completely removes the risk of SQL injections. It will be impossible to modifiy the syntactic structure of the query via a tautology. Looking back at the tautology attacks presented above, by binding the id to the already prepared statement, the search would have only resulted in a search with *id* having the string value **' OR '1'='1**, thus returning 0 records. Moreover, with prepared statements no parameter quotation is necessary, as the driver will automatically handle this.

However, this security benefit might go away if prepared statements are used in conjunction with string concatenation. Consider the next PHP example, implemented with the help of the *PDO* mysql library:

```
$stmt = $con->prepare("SELECT username, password FROM Student WHERE
                                             id=:id OR name='".$name."'");
```

```
    $stmt-> bindParam("id", $id);
    $stmt-> execute();
```

Although the *id* parameter does not pose any danger, the fact that the name is concatenated to the query string and not bound constitutes a vulnerability that might be exploited by an attacker.

Some developers disregard the idea of prepared statements because of two reasons:

- user input is restricted to take literal positions
- there is an initial performance overhead, caused by the precompiling phase

In what concerns the "only literals" downside, we suggest in section 6.4 a strategy that permits table and column names to be concatenated and white-listed for compensating the security gap.

As for the performance issue, the question is to what extent the application is affected by the precompiling step. Moreover, it would be interesting to study whether programs actually run the same prepared statement multiple times, with different parameters, so that the speed execution is improved and compensates for the precompiling time.

Our intuition is that the time overhead the precompiling phase adds is minor and should not constitute a reason to give up the security benefits that come with the use of prepared statements [13]. However, this should be further researched.

CHAPTER 3

Related work

Existing techniques targeting the identifying of SQL injection vulnerabilities and preventing exploits include defensive coding, string analysis, taint checking, runtime monitoring and test generation. Each of these approaches- falling into the category of either static, dynamic or mixed methods- comes with advantages but also limitations, offering opportunities for improvement [14]. A thourough examination of this class of research work follows in the first part of the chapter.

However, our intention is to refactor the initial source of the web applications and remove the risk of SQL injection attacks, not only mitigate the probability of accidents. In the second part of this chapter we will discuss one attempt we encountered in this direction.

3.1 Preventing SQL injection attacks

3.1.1 Defensive programming

Defensive coding includes strategies like escaping user-supplied values, data type validation, white-listing etc. We have already shown that sanitising input through exclusive manual work is an error-prone process. Developers make omissions and attackers continue to find new attack strings or variations on old attacks, circumventing their efforts [15], [14].

3.1.2 Static string analysis

Christensen et al. take the first step in statically determining the possible values of string expressions [16]. They have developed a string analysis for Java programs that approximates the set of possible string values that might result after a multitude of string manipulations. Their analysis constructs flow graphs from source files and generates a context-free grammar with a nonterminal for each string expression. The result is then widened into a regular language using a variant of an algorithm previously used for speech recognition. Next, a type of multi-level automaton is built for specific string expressions of interest in the program, addressed as ***hotspots*** [16]. With several other possible applications, their algorithm can also be used to perform static syntax checking of SQL expressions, as they have provided a regular language covering most of the common queries [16]. Hence, their work is used by a multitude of future researchers, starting from the assumption that the generated automaton is syntactically correctâĂŞ query strings are all of valid SQL syntax.

Gould et al. [17] might be considered the pioneers in the semantic checking of SQL queries [6]. They created an inter-procedural data-flow analysis that uses Christensen et al. work [16] and came up with a method to ensure that all generated queries are type-correct [17]. This is accomplished by applying a variant of the CFL reachability algorithm, first to obtain the column-name to type mapping from the schema, and next by using a type grammar for expressions to propagate type information [17]. This approach lead to some false positives due to the fact that string-analysis may over-approximate the set of possible generated strings; "precision of the string analysis could be improved, but finite state automata lack the expressive power to match precisely an arbitrary set of strings generated by a program" [18]. When this research was later re-analysed together with Gary Wassermann [18], another source of imprecision -in type checking-, was detected (rarely encountered in practice though) due to applying the CFL-reachability algorithm.

Next, Minamide [19]used some of the elements from Christensen et al. work [16] and applied it to PHP. However, instead of leveraging their analysis on regular languages, they used context-free languages, obtaining more precise approximations [19]. Moreover, Minamide modelled string operations with an automaton with output called a transducer [19] -technique from computational linguistics-, improving the precision of his analysis [8]. What Minamide succeeds in doing is however validating dynamically generated HTML pages as a whole, and this was achievable because HTML has a flatter grammar than SQL [8].

The following step in this series of studies concerns performing semantic validation of the queries to detect security vulnerabilities that would allow SQL injection attacks. Although we have the type-checking system developed by Gould et al. [17], deeper semantic analysis like tautologies' checking have not been performed, although they are the ones that constitute the real danger. Wasserman and Su [6] address this issue and again build upon the string analysis of Christensen et al. [16]. They next treat the SQL-language grammar and the generated FSA as inputs of an altered CFL-reachability algorithm and discover access control errors against the database and potential tautologies in the WHERE clauses [16].

3.1.3 Taint checking

Taint checking refers to the analysing the data flow of a program with the intent of determining whether input from an untrusted source -user input in the case of web applications- reaches a hotspot in a program, or a so-called *sensitive sink*. In the current context, this would be the execution of a query against the database.

The first relevant step in applying taint analysis to SQL injection vulnerabilities is done by Huang et al. [20], who incorporated their solution into a tool named *WEBSSARI*, targeting PHP. They "create a lattice-based static analysis algorithm derived from type systems and type-state" [20]. Because static analysis may offer unsatisfactory runtime program state, Huang et al. use static techniques to instrument sections of code potentially vulnerable with runtime guards, meant to perform sanitisation tasks [20]. Statistics show that by running the static analyzer and placing the right annotations, the potential runtime overhead is reduced by 98.4% [20].

However, WebSSARI has several key limitations that "restrict the precision and analysis power of the tool" [18]. The analysis requires three user-written "prelude" files for specifying the sensitive functions, postconditions for functions that perform sanitization and specification of all possible untrusted input providers (e.g., *$_GET, $_POST, $_REQUEST*). Moreover, since Huang et al. base their algorithm on the premise that SQL injection attacks are often the results of insecure information flow [20], we agree with Wasserman and Su [8] that real SQL injection vulnerabilities might not be detected due to missing the context of the user input, the structure of the query and also by considering the sanitization techniques safe and sound. We encounter this last assumption in another solution as well, where they use a precise points-to-analysis for Java and the query language PQL to identify program paths that allow tainted input reach the sensitive query sinks [4].

Xie and Aiken [21] expose a solution where the PHP source is parsed into abstract syntax trees (ASTs), followed by the standard conversion of the function body into a control flow graph (CFG). Next, a three-tier analysis is done that captures information at decreasing levels of granularity at the intrablock, intraprocedural and interprocedural levels, being capable to handle dynamic features of scripting languages. With Huang et al. approach [20], this was not possible. Dynamic variables written like *$$a* were considered as tainted. When dynamic PHP functions like *eval* were encountered, WebSSARI output a warning message indicating that it cannot guarantee soundness [20].

Wassermann and Su [8] use and improve upon Minamide's string analyzer. They add information flow tracking and checks on the generated grammars. They extend the analyzer with 243 PHP functions, although it still proved to have some limitations that required manual modification of the source files. Additionally, support for dynamic includes is enhanced, with the precondition that the file and directory layout are part of the specification [8].

3.1.4 Runtime techniques

As it has been previously stated, "conservative static analysis" [15] is often combined with runtime analysis, because only in this latter phase more information about data and the program state is available. We are going to present techniques that "enforce more expressive policies than simply tracking the flow of tainted input, which Perl's taint mode already provides" [17].

Halfond and Orso [15] built a tool named *AMNESIA* based on an algorithm which incorporates the existing work in string analysis([16], [17]) to statically analyse the web-application code and build a conservative model of the possible safe queries. Their intuition is that the structure of a query is entirely available in the source code and consider any attempt to alter the SQL statement to be an injection attack [15]. At runtime, following this policy, they validate all dynamically generated queries against the statically computed model.

Starting from the same premise -that all SQL injection alter the structure of the query as it was intended by the programmer- and not questioning the constant portions of the written SQL code, Buehrer et al. [22] first compute statically a parse tree, replacing user input by empty literals. Next, their algorithm compares this structure with the parse tree obtained during execution and if differences are detected, an exception is thrown. A less rigurous approach is later developed by Wassermann and Su [4], who track user's input by using meta-data -*[* and *]* delimiters are used to mark the beginning and end of each input string, building augmented queries and an augmented grammar. The only productions in which *[* and *]* occur have the form: *nonterm ::= [symbol]*, where symbol is either a terminal or a non-terminal, thus allowing input to take syntactic forms other than literals. In the solution provided by Wassermann and Su, a query is legitimate if both an input has a valid parse tree and the input's syntax is valid within the context of the query's parse tree [4].

Other runtime approaches -both Nguyen-Tuong et al. and Pietraszek and Berghe- modifiy the PHP interpreter to "track taint information at the character level, tokenize the completed query, and check whether any tainted characters appear in any tainted characters" [8]. A similar solution is provided for Java, but instead of modifying the JVM, they provide a byte code instrumenter [8].

3.1.5 Testing

In our research we have encountered a series of black-box testing and white-box testing tools for detecting SQL injection vulnerabilities. They perform by simulating real attacks and discover the security flaws. From the category of black-box testing techniques, at least four assessment frameworks for Web application security (WAVES, AppScan, WebInspect, and ScanDo) should be mentioned [20]. However, testing approaches can never guarantee soundness [20] and moreover, black-box testing is limited as it does not consider the internal representation of the application, decreasing chances for succesfully simulating a high rate of injection attacks.

White-box testing techniques are based however on the categories of approaches previously discussed, from string to static or dynamic taintedness analysis. We consider the implementation of Kiezun et al. [14] should be mentioned, that materialized into a tool named *Ardilla*, designed for PHP. This is a white-box testing tool, which creates real-attack vectors and handles dynamic programming-language constructs and can be applied on unmodified application code. It is based on dynamic taint analysis; Ardilla uses "input generation, taint propagation and input mutation to find variants of an execution that exploit a vulnerability" [14].

3.1.6 Discussion

At the beginning of the chapter we presented a series of string analysis techniques. We agree to Wassermann et al. [18] that "the string analysis is not precise and cannot be precise in general"; it is a conservative process, as formal languages are used to overestimate all the possible values a string expression can take at runtime. String analysis does not track the source of string values, so it requires the specification of regular expressions of the permitted SQL queries at each hotspot. This task falls into the responsibility of the programmer, thus being an error-prone process [18] with direct impact over the well-functioning of the defense technique.

Moreover, Christensen et al. work [16] only ensures that the generated queries are syntactically correct, without testing any of the semantics. When semantics is first adressed, in [17], it concerns type checking, whereas type violations might only lead to a database crash [15], but not a SQL injection attack. In [6] however, attacks are statically adressed, by testing for WHERE clause tautologies or access control violations.

In taint checking techniques, in the process of securing user input, functions are classified and used as either being sanitizers, or as having no effect at all over the incoming values. But as we previously argue, this evaluation is context-agnostic, though its soundness cannot be proved [8]. Moreover, tools like WEBSSARI [20] require manually-written specification files, which constitute again an error-prone technique.

As for runtime techniques, both AMNESIA [15] as also Buehrer et al. [22] base their work on the presumption that any input intended to modify the SQL query structure is an injection attack. This idea is restrictive though and completely removes the possibility of reading queries or query fragments in from a file or database [4], or simply selecting a table/ column name dynamically, via users' actions in a web interface. This limitation is overcome in the research lead by Wassermann and Su [4]. Lastly, the techniques which require the modification of the interpreter are not easily applicable to any language, because companies as Sun is unlikely to modify its Java interpreter for the sake of web applications [4].

Because vulnerabilities are known to be possible even when the above measures are taken, black-box and white-box testing tools have been built. An efficient tool has been mentioned-Ardilla-, that proved to discover unknown vulnerabilities in the projects analysed [14]. However, "while testing can be useful in practice for finding vulnerabilities, it cannot be used to make guarantees either" [6].

3.2 Removing SQL injection vulnerabilities

The safest method to remove the risk of SQL injection attacks would be by using prepared statements to separate the SQL structure from the SQL input. However, fully applying this approach has the limitation of excluding dynamically constructed query structures, as all input will take the syntactic position of literals [4], [12]. These limitations have also been encountered in Halfond and Orso [15] and Buehrer et al. [22], as previously discussed, where models of the legal queries were built and used for validation at runtime.

The work by Thomas et al. [12] proposes a prepared statement replacement algorithm (PSR-Algorithm) that traverses the source code to gather information and inserts instrumentation code in order to overcome the shortage of available information during the static analysis. Their solution is able to infer dynamic tree structures holding the SQL inputs that need to be bound to the prepared statement, maintaining the correct order in conformance with the runtime execution path. Since the tree exists at runtime in the executing code, by also placing a recursive method in the code to traverse the tree, the generation of valid prepared statements becomes possible at that point. The refactoring is achieved via the Prepared Statement Replacement Generator

(PSR-Generator), which implements the PSR-Algorithm for Java and correctly replaces 94% of the SQL injection vulnerabilities in the analysed projects [12].

The solution has a series of limitations and disadvantages however:

- the analysis of the source code is strictly based on pattern matching and does not take into consideration advanced code analysis features like call graphs or abstract syntax trees (ASTs)
- therefore, it is assumed that all non-compiled parts of the code such as comments or documentation are removed before file's conversion; moreover, what the PSR- Generator first does is formatting the source code to a standard representation so it could be further processed
- it processes one file at a time, the algorithm considering only the local variables, methods, or method calls
- the line numbers of the SQLi vulnerabilities should be first provided via an external static analyzer and accompanied by a list of guaranteed secure identifiers manually-computed
- the algorithm fully relies on code instrumentation; the study fails to analyse how many of the web applications actually require the parameters' tree to be processed dynamically

CHAPTER 4

Implementation

4.1 Support

In this section we will describe the PHP PDO library which we adopted because of its prepared statements' support and the analysis and tranformation capabilities of the Rascal meta-programming language, which we used for implementing our solution for PHP.

4.1.1 PDO library

The mysql PHP library [10] has been recently deprecated because of security flaws discovered in legacy code and two new improved extensions have been introduced with PHP 5.0. Mysqli [23] is the new variant of mysql, providing both procedural and object-oriented support. It introduced prepared statements, transactions and multiple statements execution.

The other library is PDO (PHP Data Objects) [11] and it is a actually a database abstraction layer, providing drivers for many database engines (of course including MySQL). The PDO interface puts at the programmer's disposal high-level objects for working with database connections, queries and results sets and the reason why we chose it over mysqli is because we considered the code obtained is more structured and cleaner. Moreover, mysqli functions differ syntactically by mysql only by adding an *i* in front of the now deprecated functions, therefore we cannot help questioning its future in the PHP releases.

In what concerns the code transformations, below you can see how PDO structures are used to replace some of the *mysql_* functions we refactored:

mysql

```
$con = mysql_connect(host,user,pass);
if (!$con) {
  die('Could not connect:'.mysql_error());
}

mysql_select_db(dbname, $con);


$query = mysql_query(
  "select * from T where id=".$id);


$row = mysql_fetch_row($query);
```

PDO

```
try{
  $con = new PDO('mysql:host=_;dbname=_',
    $user, $pass);
} catch (PDOException $e) {
    print "Error!:".$e->getMessage().";
    die();
}

$stmt=$con->prepare("select * from T
                          where id=?");
$stmt->bindParam(1, $id);
$stmt-> execute();
$row = $stmt->fetch();
```

As it can be seen, PDO introduced the *try/catch* statements, allowing for a more elegant database error handling mechanism. Regarding the replacement of a query with a prepared statement, the following should be noted:

- In the case of a *mysql_query* call, the connection object is not required (although it can be specified as a second argument of the call), as the last link opened by *mysql_connect()* is assumed. Preparing the statement, on the other hand, requires the explicit use of the connection object.
- The *mysql_query* call returns a query object, used afterwards for retrieving the data, whereas in the case of prepared statements, the *prepare* call is the one that produces a statement object. The statement is then used for binding parameters, query execution and data interogation.
- When it comes to binding parameters, named parameters are more clear, but for of our algorithm, unanamed placeholders fit better in the automatic process.

In case the query accepted no input, we did not use prepared statements but a PDO variant of *mysql_query*, *pdo::query*. Besides *mysql_fetch_row*, we also replaced *mysql_fetch_array*, *mysql_result*, *mysql_num_rows* and *mysql_insert_id* with their PDO equivalent forms.

4.1.2 Rascal

Rascal is a meta-programming language for code analysis and transformation, being focused on the implementation of domain-specific languages and on the rapid construction of tools for investigating and refactoring source code. Rascal provides functionality for defining grammars, parsing programs, analyzing programs code, building variants of the programs, interacting with external tools and reporting analysis results in a visual way [24].

Rascal is a statically typed language and its core contains basic data-types like booleans, integers, reals, source locations, date-time, lists, sets, tuples, maps, relations, all placed in a tree with *subtype-of* relations [25]. C and Java-like control structures are provided: *if*, *while*, *for*, *switch*, together with exception handling mechanisms [24]. Rascal is a value-oriented language, meaning that all data is immutable and new objects emerge from every applied transformation operation.

For creating more complex programs, more advanced features that “enable the full range of meta-programming capabilities“ of Rascal are present [24] [25]:

- user-defined algebraic data types (ADTs) for describing abstract syntax, as is common in functional programming languages
- a built-in grammar formalism that allows the definition of context-free grammars; the syntax is then used to generate a scannerless generalized parser to be applied in the parsing of real programming languages; via an *implode* function, the concrete syntax tree is translated into an abstract syntax tree (AST)
- advanced patern matching functionality is provided over all Rascal data types, against numbers, strings, nodes etc.; Rascal provides regular expressions matching, abstract patterns (set, list, deep match(/), negative match(!)etc.), and matching concrete syntax patterns like looping structures, also binding the required variables
- patterns can be used in multiple situations, but we mostly used them in visit statements
- visit statements' syntax is similar to that of switch statements; visiting is commonly used to traverse tree structures obtained from source code files, allowing one to match only the nodes which correspond to a certain expression or statement specification; when matching on a case has been done, arbitary code can be run, the node can be annotated with meta-information useful to the programmer or even replaced with another node of the same type

Regarding PHP programs analysis, CWI is continuously extending the functionality they provide. PHP's *duck-typing* system and the semantical differences caused by running updated PHP4 code on a PHP5 engine are some of the motivations for improving the static analytical potential of the Rascal language in this domain [24].

4.2 Proposed refactoring algorithm

The solution we implemented is able to transform mysql query calls from PHP applications into prepared statements. The algorithm performs correctly for the programs that respect a set of preconditions we came up with in our analysis and which are presented in section 4.2.1, together with a set of query patterns that derived from them.

Our prototype parses a PHP source translating it into an abstract syntax tree structure, further used for extracting query-related meta-information required by the transformation phase. This latter phase traverses the file tree structure again and uses the provided data to compute appropriate prepared statements. Before the actual replacement of the query structures is performed, we first validate the query strings against previously defined SQL grammars. Failed results are written in an error report. The strings that do validate are inserted together with their corresponding binding statements into the internal tree, which is pretty-printed back into a PHP source file. These steps are shown in the figure below:

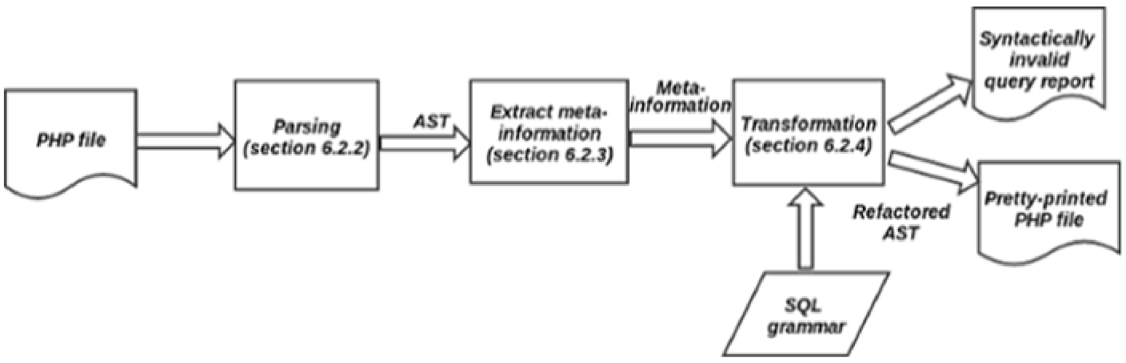
PHP file
Parsing (section 6.2.2)
AST
Extract meta-information (section 6.2.3)
Meta-information
Transformation (section 6.2.4)
Syntactically invalid query report
Pretty-printed PHP file
Refactored AST
SQL grammar

Figure 4.1: Our refactoring algorithm's main steps

4.2.1 Preconditions

Our algorithm was designed to refactor any PHP program that respects the next preconditions:

1. The implemementation is done using the *mysql_* functions.
2. Query string expressions only concatenate variables, function or method calls representing one SQL input literal.
3. All SQL structure is contained within string objects. The query string expressions are altered via assignment, concatenation or interpolation operations.
4. The statements computing the SQL query string prior to the query execution do not overlap with another query's group of statements. See below the comparison:

Consecutive query structures	**Overlapping query structures**
`$query1 = "select * from Table1";`	`$query1 = "select * from Table1";`
`mysql_query($query1);`	`$query2 = "select * from Table2";`
`$query2 = "select * from Table2";`	`mysql_query($query1);`
`mysql_query($query2);`	`mysql_query($query2);`

5. The structure of the query string expression does not depend on the program's runtime execution path

6. The query string is built and executed intraprocedural.

Based on these restrictions, we have constructed a set of representative query building patterns for which we provide the transformation theory. Their presentation is required in order to understand what we aim to achieve, before describing the prototype's method as it is reflected by 4.1.

The query execution points we used in the patterns below are wrapped in assignment statements, but our algorithm is able to handle also individual query calls and calls with the database connection specified as a second argument.

1. *mysql_query* with literal argument

   ```
   $res = mysql_query("SELECT COUNT(*) FROM students");
   ```

   Transformation: The query is safe as it does not expect any input, therefore the query execution is replaced with a simple PDO query execution.

2. *mysql_query* with literals, variables, unsafe inputs, function or method calls concatenated or interpolated

   ```
   $res = mysql_query("SELECT userid FROM students WHERE studentid = ". $id[0] .");
   ```

   A variable may only take the place of a SQL literal (precondition 2). By unsafe input, we mean a *$_POST*, *$_GET* or *$_SESSION* parameter.

   Transformation: Any concatenated or interpolated parameter is replaced with a *?* in the query string, which is then fed to the *prepare* method of the PDO connection object. The prepare, together with the binding statement(s) are inserted prior to the statement shown above. Finally, the *mysql_query* call is replaced with the execution of the prepared statement.

3. *mysql_query* with a variable as argument

   ```
   $res = mysql_query($query);
   ```

   For this form which we identified three sub-cases:

   (a) The variable gets its value from one previous assignment of either a literal or a concatenation of literals, variables or unsafe inputs.

   ```
   $query = "SELECT userid FROM students WHERE studentid = $id[0]";
   $res = mysql_query($query);
   ```

   Transformation: The first assignment is replaced with a call to the *prepare* method of the connection, with the altered query string as argument. Binding statements are inserted before the query execution, which is transformed into a prepared statement execution.

   (b) Multiple such assignments exist, distributed over *if/else* or *switch* statements.

   ```
   if ($_POST["id"] == "")
       $query = "INSERT INTO students(fname, lname) VALUES('$_POST['fname']',
                                                         '$_POST['lname']')";
   else
       $query = "UPDATE students SET fname = '$_POST['fname']',
                    lname = '$_POST['lname']' where studentid = $_POST["id"]";
       $res = mysql_query($query);
   ```

Transformation: Here, both query string assignments are replaced as in (a) and two sets of binding parameters statements are inserted under each *if* branch.

(c) The variable can modify its value across a series of assignments and append operations.

```
$query = "UPDATE students SET ";
$query .= "fname = '$fname', ";
$query .= "lname = '$lname', ";
$query .= "WHERE studentid = $id";
mysql_query($query);
```

Transformation: Because of the multiple append statements, the parameters concatenated or interpolated have to be retained in a list while traversing the set of statements, until the query execution is reached. The prepared statement is inserted before the execution call.

However, the set of append statements should not be distributed over control structures (precondition 5). If this happens, the transformation will not be statically possible because the structure of the query string and the parameters to be bound may vary with the different execution paths that the program may follow. See this reflected in the example below:

```
$query = "INSERT INTO students(";
if ($fname != "")
   $query .= "fname, lname) VALUES('$fname', '$lname')";
else
   $query .= "lname) VALUES('$lname')";
mysql_query($query);
```

4.2.2 Parsing

To parse PHP code, CWI is currently using a fork of an open-source PHP Parser [26], while efforts are made to convert an SDF parser for PHP and achieve this directly in Rascal [24]. With the functionality provided, our prototype is able to parse a PHP project recursively (given its disk location) and compute a system of file locations and their corresponding abstract syntax trees (ASTs).

The PHP AST is defined as a mutually recursive collection of Rascal datatype declarations, with base types and collection types used to represent strings, integers, lists of parameters, etc. [24]. Such a tree structure is represented in Rascal under a *Script* object, holding the list of internal statements that resides in the code. Nodes at every level are annotated with location information, from which indications like the start/end line can be extracted. The full specification of these nodes can be found in the *AbstractSyntax* module of CWI's PHP Analysis project. We present below some of these structures for a general understanding, with incomplete definitions (there are many more constructors as well as ADTs in the original module):

```
data  Script = script(list [Stmt] body) | errscript(str  err);
public  data  Stmt
  = declare(list [Declaration] decls, list [Stmt] body)
  | do(Expr cond, list [Stmt] body)
  | echo(list [Expr] exprs)
  | exprstmt(Expr expr)
  | \for(list [Expr] inits, list [Expr] conds, list [Expr] exprs, list [Stmt] body)
  | foreach(Expr arrayExpr, OptionExpr keyvar, bool  byRef, Expr asVar, list [Stmt] body)
  | function(str  name, bool  byRef, list [Param] params, list [Stmt] body)
  | \if(Expr cond, list [Stmt] body, list [ElseIf] elseIfs, OptionElse elseClause)
  | inlineHTML(str  htmlText)
  | \return(OptionExpr returnExpr)
  | \while(Expr cond, list [Stmt] body)
  | emptyStmt()
```

```
    | block(list [Stmt] body)
    ;
  public  data  Expr
    = array(list [ArrayElement] items)
    | fetchArrayDim(Expr var, OptionExpr dim)
    | assign(Expr assignTo, Expr assignExpr)
    | assignWOp(Expr assignTo, Expr assignExpr, Op operation)
    | binaryOperation(Expr left, Expr right, Op operation)
    | unaryOperation(Expr operand, Op operation)
    | eval(Expr expr)
    | exit(OptionExpr exitExpr)
    | call(NameOrExpr funName, list [ActualParameter] parameters)
    | methodCall(Expr target, NameOrExpr methodName, list [ActualParameter] parameters)
    | scalar(Scalar scalarVal)
    | var(NameOrExpr varName)
    | scriptFragment(list [Stmt] body)
    ;
```

Once the ASTs system has been obtained, we need the connection details of the application, as the connection variable will be used for preparing the future statements. The details are identified through a first visiting of the collection of scripts. The *mysql_connect* and *mysql_select_db* calls are removed at this point and replaced with a *PDO* database initialisation object, as shown in section 4.1.1. Further on, every script will be individually analysed for extracting meta-information about the query structures, which will be later used in the transformation process.

4.2.3 Extract meta-information

With this step, the intention is to traverse the *Script* object in order to obtain enough data to help the transformation process identify which refactoring method to apply for type 3 query building structures, as theoretized in section 4.2.1. Types 1 and 2 are straightforward, as there is no need to inspect the previous set of assignment and append statements.

The analysis performs on a modified version of the PHP script. With respect to precondition 4, we have seen the possibility of processing the script's statements backwards, so that once a query execution point is met, we can detect the statements that affected the query's variable argument.

The algorithm first swaps the set of statements under *if* clauses with the statements under their corresponding *else* clauses and also reverses the *cases* order in *switch* statements. This is done for the entire file. Secondly, it performs a recursive re-ordering of all control structures' body statements, as well as both functions' and top-level script's statements. The refactoring was easily performed due to Rascal's powerful visiting and matching functionality and this can be seen in the code fragment below which swaps the clauses as mentioned, based on their internal type definition. The re-ordering implementation can be found in the full code from the Appendix, section 8.2.

```
public  Script reverseStatementsInScript(Script scr) {
    scr = inverseClauses(scr);
    visit (scr) {
        case  script(_):  {
            scr.body = reverseStatements(scr.body);
        }
    };
    return  scr;
}
```

```
public  Script inverseClauses(Script scr) {
    scr = top-down  visit (scr) {
        case  ifStmt :  \if(_, _, _, someElse(_)) :  {
            aux = ifStmt.body;
            ifStmt.body = ifStmt.elseClause.e.body;
            ifStmt.elseClause.e.body = aux;
            insert  ifStmt;
        }
        case  switchStmt :  \switch(_, _) :  {
            switchStmt.cases = reverse(switchStmt.cases);
            insert  switchStmt;
        }
    };
    return  scr;
}
```

In Rascal, the patterns may bind variables in a conditional scope. A *case* statement is such a scope, thus the variables are made available to the *case*'s body. If instead, the character _ is used, this does not happen. In this example, the intention was to alter the bodies of *if* or *switch* statements, therefore we accessed the sub-elements by their internal path and did not bind them to any local variable.

Once the swaping and reversing steps have been done for the whole script, we will traverse the new structure with the intention of producing one map and ttwo lists with elements:

- For every assignment whose assignee is an argument of a run query statement, according to the query patterns 3a and 3b, we create a map pair, having the line of the assignment as a key. The tuple formed by the connection object of the corresponding query call and the variable holding the query result, becomes the key's value. Both variables in the tuple are replaced with null if they are not specified. We will call the resulted map *assigns*.
- For every assignment and append statement whose assignees are query arguments, being part of the 3c model, we retain their lines of code and build the *appends* list.
- The lines of code where *mysql_query* statements also falling into the 3c category were found compose the third list -*appendedQueries*.

We have to mention that the tranformation phase that requires these three collections will perform on the normal script. However, since we only change the statement's order and not insert new ones or delete them, the location annotations attached remain unaltered on the nodes.

Our method consists of visiting the reversed script version, matching on concrete *mysql_query* syntax patterns, as well as assignment and append statements and running a certain set of operations in order to gather the information required. There are eight Rascal patterns we used for matching on the different forms the *mysql_query* calls can take.

```
call(name(name("mysql_query" )),[actualParameter(scalar(string(_)), _)]);
call(name(name("mysql_query" )),[actualParameter(scalar(string(_)), _), _]);
call(name(name("mysql_query" )),[actualParameter(scalar(encapsed(_)),_)]);
call(name(name("mysql_query" )),[actualParameter(scalar(encapsed(_)), _),_]);
call(name(name("mysql_query" )),[actualParameter(binaryOperation(
                                                  left,right,concat()),_)]);
call(name(name("mysql_query" )),[actualParameter(binaryOperation(
                                                  left,right,concat()),_), _]);
call(name(name("mysql_query" )),[actualParameter(var(name(name(_))), _)]);
call(name(name("mysql_query" )),[actualParameter(var(name(name(_))), _), _]);
```

The second version of each pattern is required in order to address the calls which have the connection object specified as a second argument. The first two patterns are for type 1, the next four for type 2 (query string built via interpolation (*scalar(encapsed(_))*), as well as using concatenation(*binaryoperation(left, right, concat())*)) and the last two for type 3.

Before describing the prototype's behaviour once query executions, assignment and append statements have been matched (for the last two, see *assign* and *assignWOp* expression constructors in section 4.2.2), we need to introduce the notion of *query trace*. Because a query following the third pattern can only be classified into the (a), (b) or (c) sub-types after all of its statements have been processed (remember that the script we are analysing is reversed), we created a *QueryTraceVar* data structure:

- The trace is initialised with information describing the matched query call: the database connection if specified, the variable given as a parameter to the call, the line number and the variable to which the query is assigned, if it is the case. Additionally, two boolean flags and two initially empty integer lists are held in the trace. The flag *foundStartAssign* indicates whether an initial assignment statement has been encountered for the query's variable parameter. The flag *foundAppendIf* marks the finding of an append statement that follows the counterexample to (c). The two lists are the query-level versions of the lines of code in the *assigns* map, respectively the *appends* list we want to produce for the whole file.
- Because of precondition 4, one trace does never interfer with a second query's trace.
- The trace is processed at every query execution point and reinitialised, as well as at the end of the script.

We now explain the visiting algorithm:

- Whenever a query execution (type 1, 2 or 3) is matched, the query trace structure is processed in order to analyse the precedent query. If *foundStartAssign* is true (an initial assignment statement has been found, which is required in conformance with precondition 6) and elements exist in trace's *appends* list, the precedent query is classified as 3c. Its line number, stored in the trace, is appended to the *appendedQueries* list we want to produce for the file. We form map pairs from every line number in the *assigns* trace list, associated with the tuple containing the query connection variable and the variable-result of *mysql_query*. These pairs and the *appends* local trace list are appended to the final *assigns* map, respectively to the *appends* list.
- Additionally, after the trace is processed, for type 1 and 2 query executions the trace is reset, whether for type 3 it is reinitialised with the current query's information.
- Whenever an append statement is encountered whose assignee matches the query trace's variable identifier, we look at the traversal context and check whether the append is nested inside a control structure, while the query execution is not (counterexample 3c). If it is not the case, the append statement's line number is added to the *appends* list of the trace. Otherwise, we indicate such a structure has been encountered by setting the *foundAppendIf* trace flag to true.
- Whenever an assignment statement is encountered whose assignee matches to the trace, we first check whether it is a simple assignment or it actually concatenates the initial's assignee value with some new expression. If it is not the case, we mark in the trace that we found an initial assignment statement and then look at the trace's *appends* list. If this one is empty and flag *foundAppendIf* false, the statement's line number goes into the *assigns* list. Otherwise, we detected a 3c building pattern and the line number is added to the *appends* list.

  If however we discovered that the assignment actually composes the assignee with itself, the same check for control structures is done as with the append statements and we proceed accordingly.

At the end of the recursive traversal, the three collections: *assigns* map, together with the *appends* and *appendedQueries* lists are final. As we have previously mentioned, although the script was reversed, the line numbers correspond to the initial version and are ready to be processed by the next phase.

4.2.4 Transformation

This phase does the refactoring of the mysql queries into PDO prepared statements. In order to apply the transformations theoretized at the *Query patterns* section, it requires input from the inspection step -the three collections as mentioned above-.

To ease the implementation, we created a Rascal module with parametrized methods ready to build and return the AST nodes we need for the prepared statement's required structures. For example, the list of *bindParam* clauses is provided by the following utility method:

```
public  list [Stmt] bindParameters(Expr stmtExpr, list [Expr] inputs) {
    list [Stmt] clauses = [];
    int  cnt = 1;
    for  (Expr inp <;- inputs) {
        clauses += bindParam(cnt, inp, stmtExpr);
        cnt = cnt + 1;
    }
    return  clauses;
}

private  Stmt bindParam(int  offset, Expr param, Expr stmtExpr) {
    return  exprstmt(methodCall(stmtExpr, name(name("bindParam" )),
                        [actualParameter(scalar(integer(offset)),false ),
                        actualParameter(param, false )
                        ]
              ));
}
```

What our algorithm does is visiting the initial file script and matching query execution statements, as well as assignments and append statements and proceed in the following way:

- If an assignment or append statement's line number does not occur into either the *assigns* or the *appends* collections, the statement is ignored as its assignee is not the argument of any query execution call.

- If an asignment statement matched and its line number can be found in the *assigns* map, the case falls either into the 3a or 3b pattern. The statement is replaced by the corresponding *prepare* and binding calls. If actual connection and result variables were previously attached in the extraction phase to the assignment, they would be used to build the calls. Otherwise, the application's global connection is used and a generic identifier for the statement.

  If however the assignment occurs into the *appends* list, it is not replaced, but a *local prepared statement* is computed and retained as associated to the assignee, in a global map. The *PreparedStatement* data type therefore holds the *prepare* and binding calls, for future use.

  In both cases, every non-literal in the prepared statement's query string is replaced with an unknown placeholder (*?*) and the surrounding string delimiters, now redundant, are removed.

- If an append statement matched and its line number can be found in the *appends* list, another local prepared statement structure is generated and combined with the already existing prepared statement structure for the assignee. This is done by merging the query strings and concatenating the binding lists. The resulting structure is attached to the assignee, replacing the old value in the map.

- When a *mysql_query* call is encountered and falls into the first query pattern, only the replacement with the *pdo::query()* version takes place. In case of type 2, prepare and binding statements are additionally prepended. As for the third building model, if the query's line number is not included in the *appendedQueries* list, the execution call is replaced with a prepared statement execution call. However, if the line number is found, that makes it a type 3c query, with both the prepare method call and bind clauses missing. They can be found and inserted from the gradually computed prepared statement structure, associated to the query's argument.
- If database error handling is encountered via *die* functions, the structures are refactored into *try/catch* blocks.

While the file tree is being traversed, our algorithm also replaces the other mysql functions: *mysql_fetch_row*, *mysql_fetch_array*, *mysql_result*, *mysql_num_rows* and *mysql_insert_id* with their PDO equivalent.

Before the nodes with PDO structures are inserted in the AST, there is actually an extra step which validates the new prepared statement's query string against predefined SQL grammars. Except for Gould et al's approach [17] who performed type checking for the constant parts of the query, the majority of approaches we encountered "took the code as the specification" of the application [4] and did not question its syntactically corectness.

We decided to act differently, especially that Rascal supports full context-free grammars for syntax definition [25]. With the structures we implemented, we are able to parse SELECT/UPDATE/INSERT/DELETE simple commands (no table joins, unions, subqueries etc.). A successful validation would also be one of the indicators proving that our algorithm is correct in generating the query strings. The following example provides the syntax definition of the INSERT SQL statement, while the others can be found in the Appendix, section 8.3. The representations are very much based on a SQL SELECT grammar we encountered in Wassermann et al. [18].

```
module lang::php::query::\syntax::Delete

layout Standard = [\t \n \  \r \f ]*;

start syntax Delete =
    delete: "delete" "from" Table table
    | delete: "delete" "from" Table table AdditionalClauses additionalClauses
    | delete: "delete" "from" Table table WhereClause whereClause
    | delete: "delete" "from" Table table WhereClause whereClause AdditionalClauses additionalClauses;

syntax WhereClause = where: "where" Condition condition;

syntax Condition = condition: LogicTerm logicTerm
    | bracketCondition: "(" Condition condition ")"
    | notCondition: "not" LogicTerm logicTerm
    | orCondition: Condition condition "or" LogicTerm logicTerm
    | andCondition: Condition condition "and" Condition condition;

syntax LogicTerm = logicTerm: LogicFactor logicFactor
    | andTerm: LogicTerm logicTerm "and" LogicFactor logicFactor
    | bracketLogicTerm: "(" LogicTerm logicTerm ")" ;

syntax LogicFactor = comparison: Comparison comparison;

syntax Comparison = simple: ExprSimple exprLeft CompareOp compareOp ExprSimple exprRight
```

```
    | multiple:  Comparison comparison CompareOp compareOp ExprSimple exprRight
    | isNull:  ExprSimple expr "is"  "null" ;

syntax  Factor = factorColumn :  Column column
    | factorInt:  Int intVal
    | factorFloat:  Float floatVal
    | factorString:  String str
    | factorDate:  DateFunct dateFunct
    | factorExpr:  "("  ExprSimple exprSimple ")"
    | funcFactor:  Function function FuncParen funcParen;

syntax  Term =   factorTerm:  Factor factor
    | multTerm:  Term term MultOp multOp Factor factor;

syntax  ExprSimple = addExpr:  ExprSimple exprSimple AddOp addOp Term term
    | termExpr:  Term term
    | unaryExpr:  AddOp addOp Term term;

syntax  Function = upper:  "upper"
    | lower:  "lower"
    | abs:  "abs"
    | len:  "length" ;

syntax  FuncParen = funcParenExpr:  "("  ExprSimple exprSimple ")"
    | funcParenParenDbl:  "("  FuncParenDbl funcParenDbl ")" ;

syntax  FuncParenDbl = funcParenDbl:  ExprSimple exprSimple1 ","  ExprSimple exprSimple2;

syntax  AdditionalClauses = limitClause:  Limit limit
    | orderClause:  OrderBy orderBy
    | orderAndLimit:  OrderBy orderBy Limit limit;

syntax  Limit = limit:  "limit"  Int offset;

syntax  OrderBy = orderByCol:  "order"  "by"  ExprSimple expr
    | orderByColWithDirection:  "order"  "by"  ExprSimple expr OrderDirection direction;

syntax  OrderDirection = asc:  "asc"  | desc:  "desc" ;

syntax  Table = table:  Ident name
    | qtable:  "‘"  Ident name "‘" ;

syntax  Column = column:  Ident name
    | qcolumn:  "‘"  Ident name "‘"
    | tableColumn:  Ident tableName "."  Ident colName
    | qtableColumn:  "‘"  Ident tableName "."  Ident colName "‘" ;

syntax  AddOp = add:  "+"  | sub:  "-" ;

syntax  MultOp = mult:  "*"  | div:  "/" ;

syntax  CompareOp = gt:  "\>;"  | lt:  "\<;"  | eq:  "="  | ge:  "\>;="  | le:  "\<;="  |
ne:  "\<;\>;" ;

lexical  Int = [0 - 9 ]+ !>;>; [0 - 9 ];
lexical  Ident = ([a - z  A - Z  0 - 9  _ ] !<;<; [a - z  A - Z ][a - z  A - Z  0 - 9  _
]* !>;>; [a - z  A - Z  0 - 9  _ ]) | "?" ;

lexical  Float = [0 - 9 ]* "."  [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  String = "\""  StringChar* [\\ ] !<;<; "\""
| "\'"  StringChar* [\\ ] !<;<; "\'" ;

lexical  StringChar = ![\" ] | [\\ ] <;<; [\" ];

lexical  DateFunct = currdate:  "curdate()"
    | now:  "now()" ;
```

In case syntactic mistakes are found, parse errors are built indicating the file name and the line number where the broken query resides. An explanatory message is also added, but unfortunately this functionality of Rascal still needs to be improved, the indications being very vague at the moment. The final result is output into a report.

After the tree traversal is complete, the structure is pretty-printed back into a PHP file, using the *PrettyPrinter* module built by CWI's team.

CHAPTER 5

Evaluation

We have built a prototype tool able to transform PHP mysql queries into prepared statements, in order to guarantee the web applications' protection against SQL injection. Our algorithm was implemented starting from a set of preconditions detailed in section 4.2.1 that allowed us to reduce the problem space and come up with a fully static refactoring method, within the time limit we had. We then asked ourselves the following questions:

1. How many of the query building models that actually exist we managed to cover with the patterns we derived from the initial preconditions?
2. To what extension do the modern trends in PHP programming, as well as the existing dynamic features of the language affect the algorithm's transformation capabilities?
3. Is the output produced by our prototype correct?

Answers to these questions are provided throughout this chapter. We start by describing our evaluation method for each of the three issues (section 5.1), followed by the results we obtained and their discussion -section 5.2. Threats to validity are addressed in section 5.3 and in the end of the chapter we conclude whether the questions have been answered or not and how.

5.1 Evaluation method

5.1.1 Query patterns coverage

In order to work out a report for question 1, two sets of programs have been analysed and the classification of the queries has been tried in conformance to the building patterns we provided in section 4.2.1. Small modifications of the *Extract meta-information* step (section 4.2.3) were done, allowing us to obtain the following data after the visiting procedure of a reversed file script:

- the number of *mysql_query* calls with literal argument (building pattern 1)
- the number of *mysql_query* calls with literals, variables, unsafe inputs, function or method calls concatenated or interpolated (building pattern 2)
- the number of *mysql_query* calls with a variable as argument, with a single preceding assignment statement (building pattern 3a)
- the number of *mysql_query* calls with a variable as argument, with assignments distributed over control structures (building pattern 3b)

- the number of *mysql_query* calls with a variable as argument, with both assignments and appends statements, NOT distributed over control structures (building pattern 3c)
- the total number of queries in the file

We first analysed 4 open-source small projects previously inspected by Kiezun et al. [14]. The projects were downloaded from *http://sourceforge.net*: school-mate 1.5.4 (tool for school administration, 8181 lines of code, or LOC), webchess 0.9.0 (online chess game, 4722 LOC), faqforge 1.3.2 (tool for creating and managing documents, 1712 LOC) and geccbblite 0.1 (a simple bulletin board, 326 LOC). No releases/ patches have been recorded on sourceforge since 2008, although Ardilla detected for 21 SQLI vulnerabilities [14], due to the sensibility of the SQL WHERE clause. According to *sourcefourge*, schoolmate, released in 2004, stopped being updated since that year, but still had 19 downloads in the first week of October 2013.

5.1.2 Transformation limitations

We inspected a second set of programs, consisting of the projects phpBB, WordPress and phpMyAdmin which we downloaded from CWI's PHP corpus: `http://homepages.cwi.nl/~hills/experiments/corpus-icse13.tgz`. The motivation behind choosing this second sample came from the desire to strengthen our first reports by analysing larger projects, as well as searching for an answer to question 2, on the grounds that these projects are more recent and updated.

5.1.3 Evidence of correctness

Regarding question 3, we identified two possibilities. An automated, secure strategy would have validated the refactored output by running a set of test suites designed for the initial file versions. Completed by code coverage analysis, it would have constituted a complete evaluation. However, in the absence of such tests and because the limited time did not allow us to implement them ourselves, we preferred a second solution. We chose to compare the html output of the obtained php scripts with the output before the transformations. The *diff* command was used in this purpose. We have analysed 16 pages from the Schoolmate project, looking to ensure a good query coverage. Although this strategy is an incomplete evidence of our prototype's correctness, it is still a valid proof that the application's functionality is preserved.

Additionally, the query string validation we performed before the actual transformation comes in support of the correctness clause. We will present the query validation situation per the 4 small projects analysed.

5.2 Results

5.2.1 Query patterns coverage

Our analysis on the first sample lead to the following report, which is showing for every project the number of queries that fall into each building pattern we already mentioned. Comparing the number of queries classified with the total number present in the application, we discovered a small number of queries that our prototype failed to identify.

Query Patterns	Schoolmate	Webchess	Faqforge	Geccbblite	Nr of queries/pattern
Literal (1)	76	0	0	0	76
Concatenation, interpolation (2)	218	48	0	1	267
Variable-one assign (3a)	0	38	21	7	66
Variable-assigns in control flow (3b)	0	3	0	1	4
Variable-appends (3c)	0	2	11	0	13
Others	0	2	1	1	4

Table 5.1: Query report for each pattern identified in section 4.2

After manually inspecting the programs, we discovered that 3 out of the 4 queries our prototype was unable to classify were following the counterexample given for pattern 3c in section 4.2.1, presented again below:

```
$query = "INSERT INTO students(";
if ($fname != "")
    $query .= "fname, lname) VALUES('$fname', '$lname')";
else
    $query .= "lname) VALUES('$lname')";
mysql_query($query);
```

As already discussed, our tool's impossibility of transforming such structures comes from the fact that the query string can be altered in two different ways depending on the runtime execution. Therefore, the resulting sql structure cannot be statically computed.

About the fourth case, we could say it is an ambiguous structure, disregarding in one way precondition 4:

```
$query_risposte="SELECT * FROM geccBB_forum WHERE rispostadel = '$id_risposta'";
$result=mysql_query($query_risposte);
while($risp=mysql_fetch_assoc($result)) {
    [...]
    $rere=mysql_query($query_risposte);
}
```

The second query call is the one that could not be classified. The reason why this happened is because our algorithm was designed to analyse the assignment and append statements that occur before a query execution point and make use of them in order to classify the query into one of the specified patterns. However, in the above case, no assignment or append statements are found for variable *$query_risposte* after the first query execution and before the second one. This is also the reason we stated that this example can be seen as disregarding the *no overlap* precondition: the latter query call requires data defined before the previous one.

5.2.2 Transformation limitations

Our tool applies different transformation operations for the query patterns we came up with. We have seen that in the set of programs from 2004 there were just a few queries left unclassified and therefore without a refactoring solution proposed. When we analysed the second set of programs from CWI's PHP corpus, more recent and updated, with higher usage, we obtained the following results:

Query Patterns	phpBB	WordPress	phpMyAdmin	Nr of queries/pattern
Literal (1)	9	1	0	10
Concatenation, interpolation (2)	2	13	0	15
Variable-one assign (3a)	1	1	0	2
Variable-assigns in control flow (3b)	0	1	0	1
Variable-appends (3c)	1	0	0	1
Others	1	2	2	5

Table 5.2: Query report for each pattern identified in section 4.2 -CWI's PHP corpus

Although these projects are larger than the first set of applications, we found less queries than before. After performing a manual inspection, we understood that the five queries missed occur in generic query execution functions. All three frameworks run queries by calling a generic function and providing it with an already computed query string to be executed. If no assignment or append statements were encountered in the respective functions, our prototype missed classifying the query structures. In Wordpress, we actually found an asignment in this base query method, but it was disregarded because the value assigned to the query string was produced by calling an external function and altering the function's query parameter by applying some filters.

The second query we missed in Wordpress was encountered in the same aforementioned type of pattern -the counterexample to 3c-, impossible to be transformed by our static prototype.

Our intuition is that the use of one generic function to run the queries may be the "big" approach, whereas in smaller systems, people might preffer to put the query calls at the point they need them, instead of building something more complex and generic. Question 2 concerning the influence of modern trends in PHP programming over our prototype is addressed in this way.

5.2.3 Evidence of correctness

Except for the four missed queries in the analysis of the first set of programs, our prototype checked the query strings against the grammars we specified for SELECT/UPDATE/INSERT and DELETE commands. All the strings, having placeholders replacing the actual query parameters, validated successfully, except for one query which violated precondition 2: *Query string expressions only concatenate variables, function or method calls representing one SQL input literal.* The case is presented below:

```
$clause = "";
[...]
for($i=0; $i<count($semester); $i++) {
   if($i==0)
      $clause.=" AND (semesterid = $semester[$i]";
   else
      $clause.=" OR semesterid = $semester[$i]";
}
$clause.=")";
$sql = mysql_query("SELECT coursename, q1points, q2points, totalpoints,
       aperc, bperc, cperc, dperc, fperc, secondcourseid, semesterid
       FROM courses WHERE courseid = $cid[0] $clause");
```

Our prototype generated the following query string:

```
"SELECT coursename, q1points, q2points, totalpoints, aperc, bperc, cperc,
 dperc, fperc, secondcourseid, semesterid FROM courses WHERE courseid = ? ?"
```

which produced an error because the second placeholder was meant to replace a part of the sql structure.

For the scripts whose queries validated entirely we wanted to prove that pages maintained the same behavior as before the refactoring. We installed the project Schoolmate in two versions: before and after the transformation and chose 16 scripts for testing. We tried to cover as many queries as possible in every script. For example, Schoolmate follows a certain pattern by providing for each entity handled: users, students, semesters, teachers, classes etc. three scripts for:

1. displaying the necessary controls and information required to add a new entity instance
2. displaying the necessary controls and information required to edit an existing entity
3. displaying the existing records, with different filtering options and handling any insert/delete/edit post request

The third category of scripts, *Manage<entity>.php*, has at least five queries. We covered 6 such scripts and made sure to perform the operations mentioned: select, select with filtering, insert, edit and delete.

In what concerns the application's behaviour at runtime, we experienced two problems. First of all, the Schoolmate project has one file with multiple delete functions for the entities, functions that expect an id and delete the referred record. By switching to prepared statements, which require the explicit use of the connection object (as explained in section 4.1.1), a series of unidentified connection variable errors were produced because the global scope of the connection variable was not known to the functions scope. What we did to fix this was to add the database connection as an extra parameter to the delete functions, then find and modify the calls in the source code with Rascal's help.

The second problem we noticed was a filtering issue. The *ManageClasses.php* script has an *All* option for filtering the semesters. However, because the list of semesters is computed on some criteria, the *All* option's value is built in the same way:

```
$all = "";
while($semester = $query->fetch()) {
if($count == 0) {
    $all = " ".$semester["0"];
    }
    else {
        $all .= " OR semesterid = ".$semester["0"];
    }
    [...]
    $count++;
}
```

When we selected the *All* option having the value of the *$all* variable, the queries executed, in the form of:

```
SELECT COUNT(*) FROM courses WHERE semesterid = ?
SELECT c.courseid, c.semesterid, c.coursename, c.teacherid,
         c.substituteid  FROM courses c WHERE c.semesterid=?
```

violated the "only literals" precondition 2, thus breaking the prepared statement.

When we compared the html output of the initial and final html sources for the 16 scripts, the diff yielded almost the same results when used together with the xmllint tool and the –*noblanks* option to drop ignorable blank spaces. Some spaces did not get eliminated however, thus producing small differences for 6 scripts out of the 16 analysed.

5.3 Threats to validity

There are many different ways that the internal validity of our study can be threatened. Although manual analysis has been performed for all the PHP sources from the chosen projects to validate our findings, there are still chances to have missed something. Additionally, the second set of programs was only analysed but not refactored, from the consideration that we cannot provide interprocedural support.

Moreover, the queries from both sets of projects might have been classified into the wrong patterns, therefore leading to an inappropriate transformation theory. Although the pages whose html outputs we compared validated correctly, we did not analyse the majority of pages, with a broader pattern distribution. However, we think that the successful syntactic validation of the query strings increased the level of confidence into our transformations.

Furthermore, the two sets of programs did not present any dynamic PHP functionality that could have stopped our algorithm classify and perform correctly. We are wondering whether either there are other projects out there which pose these challenges when it comes to building queries, or our analysis might have missed detecting such cases and again be wrong about the transformation's correctness where this was performed, despite the query strings' successful validation.

5.4 Conclusion

We repeat the three questions we have tried to provide an answer for throughout this chapter:

1. How many of the query building models that actually exist we managed to cover with the patterns we derived from the initial preconditions?
2. To what extension do the modern trends in PHP programming, as well as the existing dynamic features of the language affect the algorithm's transformation capabilities?
3. Is the output produced by our prototype correct?

Our algorithm was able to transform a previously established set of query building models, as theorized in section 4.2.1. The coverage obtained with our predefined patterns for two sets of projects proved to be high, except for the missing interprocedural support (question 1). For Schoolmate, Webchess, Faqforge and Geccbblite all except one query string validated correctly against our SQL grammars and the transformations performed successfully. When we made a diff between the initial html outputs and the final ones for a small number of Schoolmate's pages, the results were positive (by doing this we addressed question 3).

However, our experiment presents a series of threats to validity which we discussed above. Moreover, by observing the queries our prototype missed to classify we see the need to extend our preconditions and the types of building patterns, also in conformance with question 2. A discussion follows in the next chapter.

CHAPTER 6

Improvements

In this chapter extension suggestions are discussed for our solution. We start by mentioning an instrumentation-based solution in the places where static analysis is not sufficient and bring slicing into discussion. Next, interprocedural flow handling is considered. The handling of PHP dynamic language functionalities is summarily adressed. Finally, in section 6.4, we propose a way to extend the prepared statements' constraint of enforcing SQL input to take literals positions.

6.1 Dynamic analysis

When append statements or any other type of altering statements are distributed over control structures, while the query execution point is left outside of the decisional branches, we could not provide a static algorithm for computing the query string and its corresponding binding statements. Flow sensitive static analysis would have only provided us with multiple variants of the query string, but we did not know which one to choose and use for the prepared statement until runtime. This is also why we came up with precondition 5 in section 4.2.1.

Thomas et al. [12] presented an instrumentation-based solution, their algorithm being able to infer dynamic tree structures holding the SQL inputs that need to be bound, in the right order. They insert some vector and string structures in the code, making it possible to follow the program's execution and obtain the actual runtime tree of parameters. They also place a recursive method in the PHP source to traverse the tree and generate a valid prepared statement structure to be inserted in the code and immediately executed.

We tried to apply their method but we soon realised that without a slicing algorithm that would give us the exact set of statements that flow into a certain query execution point, the amount of instrumented code becomes a problem. Since a slicing functionality for PHP was outside our thesis's scope, we chose to use CWI's Control flow graphs (CFGs) for PHP programs. These structures are built for both individual functions and methods in PHP sources, as well as for a top-level script. Besides a namepath, a CFG consists of a set of control flow nodes, like *functionEntry*, *functionExit*, *scriptEntry*,*scriptExit*, the wrapping *stmtNode* and *exprNode*s etc. The CFG also holds the edges (*FlowEdge*) set. *Label*s are used to label individual expressions and statements, with edges then going between labels.

The graph representation of a CFG consists of a set of 2-element tuples, defining a sequence of *from* and *to* nodes. What we did was finding a query execution point in the graph and then by using transitive relations we were able to find the entire set of statements that laid before that point. However, once we had this transitivity-algorithm in place, we realised that it was still completely inefficient comparing to an actual slicing algorithm, as what we had achieved was only separating an *if*'s path by its corresponding *else*'s statements.

We consider that a proper slicing algorithm, added to the static refactoring process we designed would help produce a solution with a minimum amount of instrumented code where runtime analysis is neccesary. Slicing would also help us dismiss the "no-interpolated" queries precondition, labelled with number 4. However, because of PHP's duck-typing system, operations whose dynamics change depending on the parameter's types, dynamic features like evaluation functions, includes etc., a correct and complete slicing algorithm is a very big challenge to implement.

6.2 Interprocedural analysis

Our algorithm as we designed is able to analyse and refactor query structures created and built within the same function (idea expressed in precondition 6). However, as we observed when analysing the PHP corpus assembled by CWI, large programs might use a generic function for running an already computed query string provided as an argument.

There might also be cases where query parts might be produced or altered after calling other auxiliary functions. Here too we end up knowing nothing about the actual internal representation of the query at the function's level.

In both cases, a solution we thought about whould be inlining the generic code in the function where the query is actually constructed. This would make all the information available locally and would allow static analysis. However, this could lead to an increased code size, reduced code readability and duplicated code. Performance overhead can also occur because of code duplication, however inlining can also save internal memory by removing the extra function calls from the stack.

A really straightforward solution would be firing a warning after parsing the file structure and informing the developer that refactoring into prepared statements is not achievable until interfunctional calls affecting the query are resolved.

6.3 PHP dynamic features

PHP is a highly dynamic programming language, which makes it "difficult to apply traditional static analysis techniques used in standard code analysis and transformation tools" [24]. The use of features like dynamic includes, *eval* statements with unpredictable output, variable variables etc., does not provide enough information before runtime for performing an analysis of the query structures the way we intended. In this situation, we consider that informing the developer about the code source limitations and the need for a refactoring might be the only solution for creating the possibility of transforming the queries.

6.4 Prepared statements -extension

In this section we discuss how the "only literal parameters" constraint imposed by the use of prepared statements can be extended. We consider that such an improvement might increase the popularity of parametrized queries, enlarging their dynamic potential. We suggest a solution where once a query string has been computed and parsed successfully, it could be imploded into its coresponding user-defined ADT structure. The SQL command's representation can be then visited and table and column names, for which the unknown placeholder *?* has been used, identified. The algorithm would note their position and reverse their replacement, relying again on concatenation or interpolation to attach the parameter. In order to compensate for the security flaw created, white-listing could be used to validate the table or column identifiers. We represented below these steps:

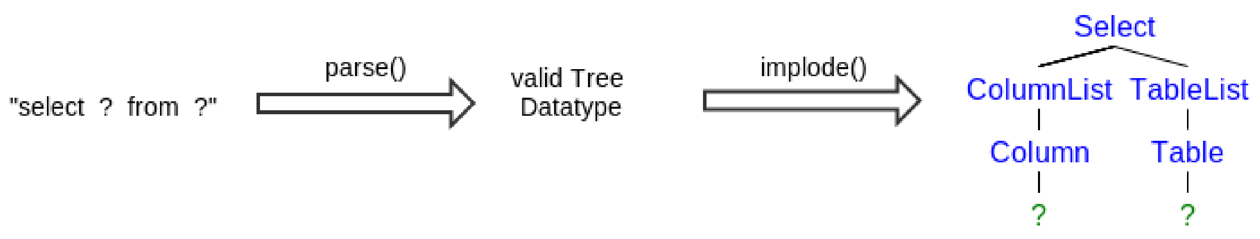

Figure 6.1: Parsing and imploding a query string into a Select ADT

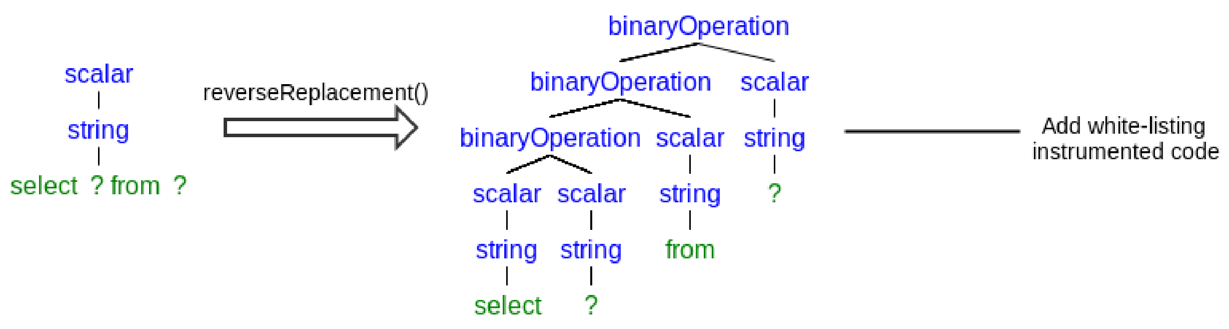

Figure 6.2: Reversing table/column names replacing

For the implementation of this process, ADTs have to be defined that correspond to the SQL grammars we have already built for Rascal. For the white-listing checks placed in the code, Rascal provides a *JDBC* module for extracting database schema-related information.

CHAPTER 7

Conclusion

We have presented an approach to completely remove the risk of SQL injection attacks from the PHP web applications. Where defensive coding proved insufficient because of its error-prone nature and automatic vulnerability detection solutions produced false positives, our prototype eliminated any chance of injection by transforming the queries into prepared statements. Although by adopting this solution the possibility of building a query with a dynamic structure is removed, we consider that the advantage of security is undoubtedly more valuable.

A set of query building patterns were established and included in our algorithm for providing a corresponding transformation strategy. On the first set of programs we analysed, the coverage of these patterns proved very high and a satisfying refactoring was performed, whereas with the second set of applications, the lack of interprocedural support showed to affect the possible transformations. We consider our solution to be a good start in the refactoring direction, but acknowledge its current limitations and suggest improvements. Extended with slicing techniques for PHP, inlining mechanisms where interprocedural interactions are encountered, the tool we have built could greatly help reducing the injection vulnerabilities encountered in most of the PHP web applications.

Bibliography

[1] Lwin Khin Shar and Hee Beng Kuan Tan. Defeating sql injection, 2013.

[2] W3techs, usage of server-side programming languages for websites, May 2013.

[3] Php-related vulnerabilities on the national vulnerability database, May 2013.

[4] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. POPL '06 Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2006.

[5] Owasp, sql-injection `https://www.owasp.org/index.php/SQL_Injection`.

[6] Gary Wassermann and Zhendong Su. An analysis framework for security in web applications. Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004), 2004.

[7] Jose Fonseca and Marco Vieira. Mapping software faults with web security vulnerabilities. International Conference on Dependable Systems and Networks: Anchorage, Alaska, 2008.

[8] Zhendong Su and Gary Wassermann. Sound and precise analysis of web applications for injection vulnerabilities. PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, 2007.

[9] P. Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. Proceedings of SCAMâĂŹ09. IEEE, 2009.

[10] Mysql api, `http://php.net/manual/en/book.mysql.php`.

[11] Php data objects, `http://php.net/manual/en/book.pdo.php`.

[12] Stephen Thomas, Laurie Williams, and Tao Xie. On automated prepared statement generation to remove sql injection vulnerabilities, 2009.

[13] Bill Karwin. Sql injection myths and fallacies presentation, 2010.

[14] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. Proceedings ICSE '09 of the 31st International Conference on Software Engineering, 2009.

[15] William G. J. Halfond and Alessandro Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. WODA '05 Proceedings of the third international workshop on Dynamic analysis, 2005.

[16] Aske Simon Christensen, Anders MÃÿller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Proc. 10th International Static Analysis Symposium, SAS âĂŹ03, 2003.

[17] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. ICSE '04 Proceedings of the 26th International Conference on Software Engineering, 2004.

[18] Gary Wassermann, Carl Gould, Zhendong Su, and Premukumar Devanbu. Static checking of dynamically generated queries in database applications, 2007.

[19] Yasuhiko Minamide. Static approximation of dynamically generated web. Proceedings of WWW 2005, ACM, 2005.

[20] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. WWW '04 Proceedings of the 13th international conference on World Wide Web, 2004.

[21] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. USENIX-SS'06 Proceedings of the 15th conference on USENIX Security Symposium, 2006.

[22] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC, 2005.

[23] Mysql improved extension, `http://php.net/manual/en/book.mysqli.php`.

[24] Mark Hills, Paul Klint, and Jurgen J. Vinju. Program analysis scenarios in rascal. WRLA'12 Proceedings of the 9th international conference on Rewriting Logic and Its Applications, 2012.

[25] Rascal tutor, `http://tutor.rascal-mpl.org/`.

[26] Phpparser, `https://github.com/nikic/PHP-Parser`.

CHAPTER 8

Appendix

8.1 Vulnerabilities analysed

Web-application	Versions analyzed	#Vuln.
PHP-Nuke	6.0,6.5,6.9,7.0,7.2,7.6,7.7, 7.8, 7.9	295
Drupal	4.5.5, 4.5.6, 4.6.5, 4.6.6, 4.6.7,4.6.8, 4.6.9, 4.6.10, 4.6.11, 4.7.6, 5.1	59
PHP-Fusion	6.00.106, 6.00.108, 6.00.110, 6.00.204, 6.00.206, 6.00.207, 6.00.303,6.00.304, 6.01.4, 6.01.5, 6.01.6, 6.01.7, 6.01.8,6.01.9,6.01.10, 6.01.11,6.01.12	54
Wordpress	1.2.1, 1.2.2, 1.5.2-1, 2.0, 2.0.10-RC2, 2.0.4, 2.0.5, 2.0.6, 2.1.2, 2.1.3 2.1.3- RC2, 2.2, 2.2.1, 2.3	115
phpMyAdmin	2.1.10, 2.4.0, 2.5.2, 2.5.6, 2.5.7PLl, 2.6.3PLl, 2.6.4, 2.6.4PL4, 2.7.0PL2, 2.8.2.4, 2.9.0,2.9.1.1,2.10.0.2, 2.1 0.1, 2.11.1.1, 2.11.1.2 and SVN revisions	74
phpBB	2.0.3, 2.0.5, 2.0.6, 2.0.6c, 2.0.7, 2.0.8, 2.0.9, 2.0.10, 2.0.16, 2.0.17	58

Table 8.1: Versions of the web application used and number of vulnerabilities analyzed [7]

8.2 Extraction

ReverseStatements.rsc

```
module lang::php::query::refactor::util::ReverseStatements

import IO;
import String;
import lang::php::util::Utils;
import lang::php::ast::AbstractSyntax;
import  lang::php::util::System;
import  List;
import  Map;
import  Node;

anno  bool  Stmt @ visited;
anno  bool  Else @ visited;
anno  bool  Case @ visited;
anno  bool  ElseIf @ visited;

public  Script reverseStatementsInScript(Script scr) {
    scr = inverseClauses(scr);
    visit (scr) {
        case  script(_):  {
            scr.body = reverseStatements(scr.body);
        }
    };
    return  scr;
}

public  Script inverseClauses(Script scr) {
    scr = top-down  visit (scr) {
        case  ifStmt :  \if(_, _, _, someElse(_)) :  {
            aux = ifStmt.body;
            ifStmt.body = ifStmt.elseClause.e.body;
            ifStmt.elseClause.e.body = aux;
            insert  ifStmt;
        }
        case  switchStmt :  \switch(_, _) :  {
            switchStmt.cases = reverse(switchStmt.cases);
            insert  switchStmt;
        }
    };
    return  scr;
}

public  Stmt reversedStmt(Stmt stmt) {
    stmt.body = reverseStatements(stmt.body);
    stmt@visited = true ;
    return  stmt;
}

public  Else reversedElse(Else stmt) {
    stmt.body = reverseStatements(stmt.body);
    stmt@visited = true ;
    return  stmt;
}

public  ElseIf reversedElseIf(ElseIf stmt) {
    stmt.body = reverseStatements(stmt.body);
    stmt@visited = true ;
    return  stmt;
}

public  Case reversedCase(Case stmt) {
    stmt.body = reverseStatements(stmt.body);
    stmt@visited = true ;
    return  stmt;
}

public  list [Stmt] reverseStatements(list [Stmt] statements) {
    statements = top-down  visit (statements) {
        case  ifStmt :  \if(_, _, _, _) :  {
            if  ("visited"  in  getAnnotations(ifStmt)) {
                    fail  statements;
            }
            insert  reversedStmt(ifStmt);
        }
        case  elseStmt :  \else(_):  {
            if  ("visited"  in  getAnnotations(elseStmt)) {
                fail  statements;
            }
            insert  reversedElse(elseStmt);
        }
        case  doStmt:  do(_, _):  {
            if  ("visited"  in  getAnnotations(doStmt)) {
                fail  statements;
            }
            insert  reversedStmt(doStmt);
        }
        case  forStmt:  \for(_, _, _, _):  {
            if  ("visited"  in  getAnnotations(forStmt)) {
                fail  statements;
            }
            insert  reversedStmt(forStmt);
        }
        case  foreachStmt:  foreach(_, _, _, _, _):  {
            if  ("visited"  in  getAnnotations(foreachStmt)) {
                fail  statements;
            }
            insert  reversedStmt(foreachStmt);
        }
        case  functionStmt:  function(_, _, _, _):  {
            if  ("visited"  in  getAnnotations(functionStmt)) {
                fail  statements;
            }
            insert  reversedStmt(functionStmt);
        }
        case  caseStmt :  \case(_, _) :  {
            if  ("visited"  in  getAnnotations(caseStmt)) {
                fail  statements;
            }
            insert  reversedCase(caseStmt);
        }
        case  elseIfStmt :  elseIf(_, _):  {
            if  ("visited"  in  getAnnotations(elseIfStmt)) {
                fail  statements;
            }
            insert  reversedElseIf(elseIfStmt);
        }
        case  whileStmt :    \while(_, _) :  {
            if  ("visited"  in  getAnnotations(whileStmt)) {
                fail  statements;
            }
            insert  reversedStmt(whileStmt);
        }
        case  blockStmt :  block(body) :  {
            if  ("visited"  in  getAnnotations(blockStmt)) {
                fail  statements;
            }
            insert  reversedStmt(blockStmt);
        }
    };
    return  reverse(statements);
}
```

QueryInspect.rsc

```
module  lang::php::query::refactor::QueryInspect

import  IO;
import  String;
import  lang::php::util::Utils;
import  lang::php::ast::AbstractSyntax;
import  lang::php::util::System;
import  List;
import  Map;
import  Node;
import  Traversal;
import  util::Maybe;
import  lang::php::query::refactor::util::ReverseStatements;
import  lang::php::query::refactor::util::AnalyzeExpr;
import  lang::php::query::refactor::QueryReport;
import  lang::php::query::refactor::QueryInfo;

data  QueryVarTrace = qtrace(QueryInfo info, bool  foundAssign, list [int ] assigns,
    list [int ] appends, bool  foundAppendIf);

QueryVarTrace resetTrace() {
    list [int ] intList = [];
    return  qtrace(queryDescr(), false , intList, intList, false );
}

QueryVarTrace resetTraceFlags(QueryVarTrace trace) {
    list [int ] intList = [];
    return  qtrace(trace.info, false , intList, intList, false );
}

map [int , tuple [Maybe[Expr] con, Maybe[Expr] result]] addToAssigns(
    map [int , tuple [Maybe[Expr] con, Maybe[Expr] result]] globalAssigns,
    QueryVarTrace trace) {
    tuple [Maybe[Expr] con, Maybe[Expr] result] aTuple =
        <trace.info.con, trace.info.result>;
    for  (int  line <;- trace.assigns) {
        globalAssigns[line] = aTuple;
    }
    return  globalAssigns;
}
```

```
public  QueryReport extractQueryInformation(Script scr, loc  l) {
    scr = reverseStatementsInScript(scr);
    int  cntqrun = 0 ;
    list [int ] varWOpLines = [];
    map [int , tuple [Maybe[Expr] con, Maybe[Expr] result]] assigns = ();
    list [int ] appends = [];
    QueryVarTrace trace = resetTrace();
    top-down  visit (scr) {
        case  qrun :  call(name(name("mysql_query" )), _) :  {
            cntqrun = cntqrun + 1;
        }
    };
    if  (cntqrun == 0 ){
        return  report();
    }
    scr = top-down  visit (scr) {
        case  qrun :  call(name(name("mysql_query" )), params):  {
            if  (![actualParameter(scalar(string(_)), _)] := params && ![actualParameter(scalar
                (string(_)), _), _] := params && ![actualParameter(scalar(encapsed(_)),
                _)] := params && ![actualParameter(scalar(encapsed(_)), _), _] := params
                && ![actualParameter(binaryOperation(left, right, concat()), _)] := params
                && ![actualParameter(binaryOperation(left, right, concat()), _), _] := params)
            {
                fail  scr;
            }
            if  (trace.info == queryDescr()) {
                fail  scr;
            }
            if  (trace.foundAssign && size(trace.appends) >; 0 ) {
                varWOpLines += trace.info.line;
            }
            appends += trace.appends;
            assigns = addToAssigns(assigns, trace);
            trace = resetTrace();
        }
        case  qrunVar:  call(name(name("mysql_query" )), [actualParameter(var(name(name
            (varName))), _)]):  {
            if  (trace.info == queryDescr()) {
                trace.info = queryDescr(nothing(), qrunVar.parameters[0 ].expr, qrunVar@at.begin.line,
                      getQueryResultVariable(qrunVar));
                trace = resetTraceFlags(trace);
                fail  scr;
            }
            if  (trace.foundAssign && size(trace.appends) >; 0 ) {
                varWOpLines += trace.info.line;
            }
            appends += trace.appends;
            assigns = addToAssigns(assigns, trace);
            trace.info = queryDescr(nothing(), qrunVar.parameters[0 ].expr, qrunVar@at.begin.line,
                      getQueryResultVariable(qrunVar));
            trace = resetTraceFlags(trace);
        }
        //With connection argument
        case  qrunVar:  call(name(name("mysql_query" )), [actualParameter(var(name(name
            (varName))), _), _]):  {
```

```
            if  (trace.info == queryDescr()) {
                trace.info = queryDescr(just(qrunVar.parameters[1].expr), qrunVar.parameters[
                      0 ].expr,qrunVar@at.begin.line,getQueryResultVariable(qrunVar));
                trace = resetTraceFlags(trace);
                fail  scr;
            }
            if  (trace.foundAssign && size(trace.appends) >; 0 ) {
                varWOpLines += trace.info.line;
            }
            appends += trace.appends;
            assigns = addToAssigns(assigns, trace);
            trace.info = queryDescr(just(qrunVar.parameters[1].expr), qrunVar.parameters[
                     0 ].expr,qrunVar@at.begin.line,getQueryResultVariable(qrunVar));
            trace = resetTraceFlags(trace);
        }
        case  assignqVar :  exprstmt(assign(var(name(name(varName))), rightAssign)):  {
            if  (trace.info != queryDescr() && varName != trace.info.queryParam.varName.name.name)
            {
                fail  scr;
            }
            bool  isVarComposedWithItself = isVarComposedWithItself(rightAssign, varName);
            if  (!isVarComposedWithItself) {
                if  (size(trace.appends) >; 0 ) {
                    trace.appends += [assignqVar@at.begin.line];
                }
                else  if  (!trace.foundAppendIf) {
                    trace.assigns += [assignqVar@at.begin.line];
                }
                trace.foundAssign = true ;
                fail  scr;
            }
            if  (!isAppendIfCase(trace.info.line)) {
                trace.appends += [assignqVar@at.begin.line];
            }
            else  {
                trace.foundAppendIf = true ;
            }
          }
       case  appendqVar:exprstmt(assignWOp(var(name(name(varName))),_,concat())):
            {
            if (trace.info != queryDescr() && varName != trace.info.queryParam.varName.name.name)
{
                fail  scr;
            }
            if  (!isAppendIfCase(trace.info.line)) {
                trace.appends += [appendqVar@at.begin.line];
            }
            else  {
                trace.foundAppendIf = true ;
            }
       }
    };
    if  (trace.foundAssign && size(trace.appends) >; 0 ) {
        varWOpLines += trace.info.line;
    }
    if  (trace.info != queryDescr()) {
        appends += trace.appends;
        assigns = addToAssigns(assigns, trace);
    }
    QueryReport report = report(varWOpLines, assigns, appends);
    return  report;
}

public  bool  isAppendIfCase(int  queryLine) {
    context = getTraversalContext()[2];
    top-down  visit (context) {
        case  ifStmt:  \if(_, _, _, _):  {
            if  (context != ifStmt) {
                return  false ;
            }
            if ((ifStmt@at.begin.line <;= queryLine) &&
                (ifStmt@at.end.line >;= queryLine)) {
                return  false ;
            }
            return  true ;
        }
        case  elseClause:  \else(_):  {
            if  (context != elseClause) {
                return  false ;
            }
            if ((elseClause@at.begin.line <;= queryLine) &&
                (elseClause@at.end.line >;= queryLine)) {
                return  false ;
            }
            return  true ;
         }
    };
    return  false ;
}

public  Maybe[Expr] getQueryResultVariable(Expr queryCall) {
    context = getTraversalContext()[1];
    top-down  visit (context) {
        case  anAssign:  assign(var(name(name(_))), call(name(name("mysql_query" )), _)):
        {
            if  (context == anAssign) {
                 return  just(anAssign.assignTo);
            }
        }
    };
    return  nothing();
}
```

8.3 Transformation

PreparedStatement.rsc

```
module  lang::php::query::refactor::PreparedStatement
```

```
import lang::php::ast::AbstractSyntax;
import lang::php::query::refactor::util::AnalyzeExpr;
import IO;
import Traversal;
import List;

public data PreparedStatement =
    prepStat(str queryString, list[Expr] inputs);

public PreparedStatement concatenatePreparedStructure(PreparedStatement prep1,
                                                      PreparedStatement prep2) {
    return prepStat(prep1.queryString + prep2.queryString, prep1.inputs + prep2.inputs);
}

public PreparedStatement extractFromExpr(Expr queryExpr) {
    list[Expr] inputs = [];
    queryStr = top-down-break visit (queryExpr) {
        case functionCall : call(_, _): {
            inputs += functionCall;
            insert scalar(string("?"));
        }
        case ternaryExpr: ternary(_, _, _): {
            inputs += ternaryExpr;
            insert scalar(string("?"));
        }
        case arrayVar : fetchArrayDim(var(name(name(arrayName))),
                              someExpr(requestParam)): {
            inputs += arrayVar;
            insert scalar(string("?"));
        }
        case aVar : var(name(name(varName))) : {
            if (varName == "_POST" || varName == "_GET" || varName == "_SESSION") {
                fail queryStr;
            }
            bool isArrayVar = false;
            if (size(getTraversalContext())>= 2 && fetchArrayDim(_,_) :=getTraversalContext()[1]){
                isArrayVar = true;
            }
            if (size(getTraversalContext())>= 3 && fetchArrayDim(_,_) :=getTraversalContext()[2]){
                isArrayVar = true;
            }
            if (isArrayVar) {
                fail queryStr;
            }
            inputs += aVar;
            insert scalar(string("?"));
       }
    };
    return prepStat(getExprStringValue(queryStr), inputs);
}
```

ASTQueryUtil.rsc

```
module lang::php::query::refactor::util::ASTQueryUtil

import lang::php::ast::AbstractSyntax;
import  lang::php::query::refactor::Connection;
import  util::Maybe;

public  Stmt prepare(Expr con, Expr queryString, Expr stmtExpr) {
    return  exprstmt(assign(stmtExpr,
                            methodCall(con, name(name("prepare" )),
                            [actualParameter(queryString, false )]
                            )));
}

public  list [Stmt] bindParameters(Expr stmtExpr, list [Expr] inputs) {
    list [Stmt] clauses = [];
    int  cnt = 1;
    for  (Expr inp <;- inputs) {
        clauses += bindParam(cnt, inp, stmtExpr);
        cnt = cnt + 1;
    }
    return  clauses;
}

private  Stmt bindParam(int  offset, Expr param, Expr stmtExpr) {
    return  exprstmt(methodCall(stmtExpr, name(name("bindParam" )),
                        [actualParameter(scalar(integer(offset)),false ),
                        actualParameter(param, false )
                        ]
                  ));
}

public  Expr executePrepExpr(Expr stmtExpr) {
    return  methodCall(stmtExpr, name(name("execute" )), []);
}

public  Stmt executePrep(Expr stmtExpr) {
    return  exprstmt(executePrepExpr(stmtExpr));
}

public  Stmt query(Expr con, Expr queryString, Expr stmtExpr) {
    return  exprstmt(queryExpr(con, queryString, stmtExpr));
}

public  Expr queryExpr(Expr con, Expr queryString, Expr stmtExpr) {
    return  assign(stmtExpr, methodCall(con, name(name("query" )),
                            [actualParameter(queryString, false )]
                            ));
}

public  Stmt fetchAll(Expr stmtExpr) {
    return  exprstmt(methodCall(stmtExpr, name(name("fetchAll" )), []));
}

public  Stmt fetchAllAndReturn(str  resultId, Expr stmtExpr) {
    return  exprstmt(assign(var(name(name(resultId))),
            methodCall(stmtExpr, name(name("fetchAll" )), [])));
}

public  Stmt dieWithError(Maybe[Expr] errorMessage) {
    if  (errorMessage == nothing()) {
         return  exprstmt(exit(someExpr(
                 methodCall(var(name(name("e" ))),name(name("getMessage" )),[])
                 )));
    }
    return  exprstmt(exit(someExpr(binaryOperation(
                                    errorMessage.val,
                                    methodCall(var(name(name("e" ))),name(name("getMessage"
                                    )),[]),concat())))));
}
```

QueryTransformation.rsc

```
module  lang::php::query::refactor::QueryTransform

import  lang::php::query::util::Parse;
import  IO;
import  String;
import  lang::php::util::Utils;
import  lang::php::ast::AbstractSyntax;
import  lang::php::util::System;
import  lang::php::pp::PrettyPrinter;
import  List;
import  Map;
import  Node;
import  Traversal;
import  lang::php::query::refactor::PreparedStatement;
import  lang::php::query::refactor::util::ASTQueryUtil;
import  lang::php::query::refactor::ProcessConnection;
import  lang::php::query::refactor::QueryInspect;
import  lang::php::query::refactor::QueryReport;
import  lang::php::query::refactor::util::AnalyzeExpr;
import  lang::php::query::refactor::QueryInfo;
import  util::Maybe;

public  list [Stmt] getStatements(QueryInfo queryInfo,
                         map [Expr, PreparedStatement] prepareInfo) {

    Expr stmtExpr = (queryInfo.result == nothing() ?  var(name(name("stmt" ))) :  queryInfo.result.val);
    list [Stmt] statements = [];
    if  (var(name(name(/\w/))) := queryInfo.queryParam) {

        PreparedStatement prep = prepareInfo[queryInfo.queryParam];
        return  writePrepareStatementWithParams(prep, queryInfo.con.val, stmtExpr);
    }
    if  (binaryOperation(_, _, concat()) := queryInfo.queryParam ||
             scalar(encapsed(_)) := queryInfo.queryParam) {
        return  extractAndWritePrepareStatementWithParams(
             queryInfo.queryParam, queryInfo.con.val, stmtExpr);
    }
    return  statements;
}

public  Maybe[Expr] newQueryExecuteCall(QueryInfo query,
             list [int ] queryrunappendLines) {
    Expr stmtExpr = (query.result == nothing() ?  var(name(name("stmt" ))) :  query.result.val);
    bool  isInAppendLines = query.line in  queryrunappendLines;
    if  (var(name(name(/\w/))) := query.queryParam && !isInAppendLines) {
        return  just(executePrepExpr(stmtExpr));
    }
    if  (scalar(string(/\w/)) := query.queryParam) {
        try {
            parse(query.queryParam.scalarVal.strVal);
        }
        catch  ParseError(loc  l):  {
            println("Failed parse <; l>;" );
        }
        return  just(queryExpr(query.con.val, query.queryParam, stmtExpr));
    }
    return  nothing();
}

public  Script transform(Script scr, Expr con, QueryReport qreport) {
    Expr stmtExpr = var(name(name("stmt" )));
    list [int ] queryrunappendLines = getqappendLines(qreport);
    map [Expr, PreparedStatement] prepareInfo = ();
    scr = top-down  visit (scr) {
    case  fetchRow:  assign(rowVar, call(name(name("mysql_fetch_array" )), [actualParameter
            (param, false )])):  {
            list [ActualParameter] params = [];
            insert  assign(rowVar, methodCall(param, name(name("fetch" )), params));
        }
        case  fetchRow:  assign(rowVar, call(name(name("mysql_fetch_row" )), [actualParameter
            (param, false )])):  {
            list [ActualParameter] params = [];
            insert  assign(rowVar, methodCall(param, name(name("fetch" )), params));
        }
        case  fetchFirstRowColumn:  call(name(name("mysql_result" )), [actualParameter(param,
            false ), actualParameter(offset, false )]):  {
            list [ActualParameter] params = [];
            insert  methodCall(param, name(name("fetch" )), [actualParameter(fetchConst(name(
            "PDO::FETCH_COLUMN" )), false )]);
        }
        case  countResults:  call(name(name("mysql_num_rows" )), [actualParameter(param,
            false )]):  {
            list [ActualParameter] params = [];
            insert  methodCall(param, name(name("rowCount" )), params);
        }
        case  lastId:  call(name(name("mysql_insert_id" )), _):  {
            list [ActualParameter] params = [];
            insert  methodCall(con, name(name("lastInsertId" )), params);
        }
        //assign
        case  queryRun :  exprstmt(assign(result,
                call(name(name("mysql_query" )), actualParameters))):  {
            Expr localCon = (size(actualParameters)==2)?actualParameters[1].expr:con;
            QueryInfo query = queryDescr(just(localCon), actualParameters[0 ].expr,
                queryRun@at.begin.line, just(result));
            Maybe[Expr] execute = newQueryExecuteCall(query, queryrunappendLines);
            if (execute != nothing()) {
                insert  exprstmt(execute.val);
            }
            list [Stmt] statements =
                getStatements(query, prepareInfo);
            if  (var(name(name(/\w/))) := actualParameters[0 ].expr &&
                actualParameters[0 ].expr in  prepareInfo) {
                delete(prepareInfo, actualParameters[0 ].expr);
            }
            insert  block(statements + executePrep(result));
        }
        //no assign
        case  queryRun :  exprstmt(call(name(name("mysql_query" )),
                actualParameters)):  {
            Expr localCon = (size(actualParameters)==2)?actualParameters[1].expr:con;
            QueryInfo query = queryDescr(just(localCon), actualParameters[0 ].expr,
                queryRun@at.begin.line, nothing());
            Maybe[Expr] execute = newQueryExecuteCall(query, queryrunappendLines);
            if (execute != nothing()) {
                 insert  exprstmt(execute.val);
            }
            list [Stmt] statements =
                getStatements(query, prepareInfo);
            if  (var(name(name(/\w/))) := actualParameters[0 ].expr &&
                actualParameters[0 ].expr in  prepareInfo) {
                delete(prepareInfo, actualParameters[0 ].expr);
            }
            insert  block(statements + executePrep(stmtExpr));
        }
        //run or die with assign
        case  queryRun :  exprstmt(binaryOperation(assign(result,
             call(name(name("mysql_query" )), actualParameters)),
             exit(someExpr(exitInfo)), logicalOr())):  {
            list [Stmt] elseStatements = [];
            if  (binaryOperation(_,call(name(name("mysql_error" )),[]), concat()) := exitInfo)
            {
                  elseStatements = [dieWithError(just(exitInfo.left))];
            }
            if  (call(name(name("mysql_error" )),[]) := exitInfo) {
                  elseStatements = [dieWithError(nothing())];
            }
            Expr localCon =(size(actualParameters) == 2)?actualParameters[1].expr:con;
            QueryInfo query = queryDescr(just(localCon), actualParameters[0 ].expr,
                  queryRun@at.begin.line, just(result));
            Maybe[Expr] execute = newQueryExecuteCall(query, queryrunappendLines);
            if (execute != nothing()) {
                insert  \tryCatch([exprstmt(execute.val)], [\catch(name(
                   "PDOException" ), "$e" , elseStatements)]);
            }
            list [Stmt] statements = getStatements(query, prepareInfo);
            if  (var(name(name(/\w/))) := actualParameters[0 ].expr &&
                actualParameters[0 ].expr in  prepareInfo) {
                delete(prepareInfo, actualParameters[0 ].expr);
            }
            insert  \tryCatch(statements + executePrep(result), [\catch(name(
                "PDOException" ), "$e" , elseStatements)]);
        }
        //run or die no assign
        case  queryRun :  exprstmt(binaryOperation(
                    call(name(name("mysql_query" )), actualParameters),
                    exit(someExpr(exitInfo)), logicalOr())):  {
            list [Stmt] elseStatements = [];
            if  (binaryOperation(_,call(name(name("mysql_error" )),[]), concat()) := exitInfo)
            {
                 elseStatements = [dieWithError(just(exitInfo.left))];
            }
            if  (call(name(name("mysql_error" )),[]) := exitInfo) {
                elseStatements = [dieWithError(nothing())];
            }
            Expr localCon =(size(actualParameters)==2)?actualParameters[1].expr:con;
            QueryInfo query = queryDescr(just(localCon), actualParameters[0 ].expr,
                queryRun@at.begin.line, nothing());
            Maybe[Expr] execute = newQueryExecuteCall(query, queryrunappendLines);
            if (execute != nothing()) {
                  insert  \tryCatch([exprstmt(execute.val)], [\catch(name(
                      "PDOException" ), "$e" , elseStatements)]);
            }
            list [Stmt] statements = getStatements(query, prepareInfo);
            if  (var(name(name(/\w/))) := actualParameters[0 ].expr &&
                actualParameters[0 ].expr in  prepareInfo) {
                delete(prepareInfo, actualParameters[0 ].expr);
            }
            insert  \tryCatch(statements + executePrep(stmtExpr),
                [\catch(name("PDOException" ), "$e" , elseStatements)]);
        }
        case  assign :  exprstmt(assign(assignee, assignedExpr)):  {
            if  ("at"  notin  getAnnotations(assign) || qreport == report()) {
                fail  scr;
            }
            int  line = assign@at.begin.line;
            if  (line notin  qreport.assigns && line notin  qreport.appends) {
                fail  scr;
            }
            if  (line in  qreport.assigns) {
                Expr queryCon =
                   (qreport.assigns[line].con == nothing()) ?  con :  qreport.assigns[line].con.val;
                Expr queryStmtExpr =
                   (qreport.assigns[line].result == nothing()) ?  stmtExpr :  qreport.assigns[line].result.val;
                insert  block(extractAndWritePrepareStatementWithParams(assignedExpr, queryCon,
                   queryStmtExpr));
            }
            if  (line in  qreport.appends) {
                prepareInfo[assignee] = extractFromExpr(assignedExpr);
            }
        }
        case  appendq :  exprstmt(assignWOp(assignee, assignedExpr, concat())) :  {
            if  ("at"  notin  getAnnotations(appendq) || qreport == report() ||
                appendq@at.begin.line notin  qreport.appends) {
                fail  scr;
            }
            prepareInfo[assignee] = concatenatePreparedStructure(prepareInfo[assignee],
                extractFromExpr(assignedExpr));
        }
    };
    return  scr;
```

```
}

public  list [Stmt] extractAndWritePrepareStatementWithParams(
        Expr queryStringExpr, Expr con, Expr stmtExpr) throws  ParseError {
    PreparedStatement stmt = extractFromExpr(queryStringExpr);
    return  writePrepareStatementWithParams(stmt, con, stmtExpr);
}

public  list [Stmt] writePrepareStatementWithParams(PreparedStatement prep, Expr con,
        Expr stmtExpr) throws  ParseError {
    list [Stmt] toInsert = [];
    try  {
        parse(prep.queryString);
    }
    catch  ParseError(loc  l):  {
        println("Failed parse <; l>;" );
    }
    toInsert += prepare(con, scalar(string(prep.queryString)), stmtExpr);
    toInsert += bindParameters(stmtExpr, prep.inputs);
    return  toInsert;
}
```

Select.rsc

```
module  lang::php::query::\syntax::Select

layout  Standard = [\t \n \  \r \f ]*;

start  syntax  Select =
    select:  "select"  ColumnList columnList "from"  TableList tableList
    | selectDistinct:  "select"  "distinct"  ColumnList columnList "from"  TableList tableList
    | selectAll:  "select"  "all"  ColumnList columnList "from"  TableList tableList
    | select:  "select"  ColumnList columnList "from"  TableList tableList WhereClause whereClause
    | selectDistinct:  "select"  "distinct"  ColumnList columnList "from"  TableList tableList
WhereClause whereClause
    | selectAll:  "select"  "all"  ColumnList columnList "from"  TableList tableList WhereClause
whereClause
    | select:  "select"  ColumnList columnList "from"  TableList tableList AdditionalClauses
additionalClauses
    | selectDistinct:  "select"  "distinct"  ColumnList columnList "from"  TableList tableList
AdditionalClauses additionalClauses
    | selectAll:  "select"  "all"  ColumnList columnList "from"  TableList tableList AdditionalClauses
additionalClauses
    | select:  "select"  ColumnList columnList "from"  TableList tableList WhereClause whereClause
AdditionalClauses additionalClauses
    | selectDistinct:  "select"  "distinct"  ColumnList columnList "from"  TableList tableList
WhereClause whereClause AdditionalClauses additionalClauses
    | selectAll:  "select"  "all"  ColumnList columnList "from"  TableList tableList WhereClause
whereClause AdditionalClauses additionalClauses;

syntax  Subquery = subquery:  "("  Select select ")" ;

syntax  ColumnList = columns:  {Column "," }+ columns
    | star:  "*" ;

syntax  TableList = tables:  {Table "," }+ tables
```

```
    | joinedTables:  Join join;

syntax  Join =
    innerJoin:  Table table1 "inner join"  Table table2 "on"  ExprSimple expr1 "="
ExprSimple expr2
    | leftJoin:  Table table1 "left join"  Table table2 "on"  ExprSimple expr1 "="
ExprSimple expr2
    | rightJoin:  Table table1 "right join"  Table table2 "on"  ExprSimple expr1 "="
ExprSimple expr2;

syntax  Column = column:  ExprSimple exprSimple
    | columnAs:  ExprSimple exprSimple "as"  Ident name;

syntax  Table = table:  Ident name
    | tableAlias:  Ident tableName Ident tableAlias
    | tableAs:  Ident tableName "as"  Ident tableAlias;

syntax  ExprSimple = addExpr:  ExprSimple exprSimple AddOp addOp Term term
    | termExpr:  Term term
    | unaryExpr:  AddOp addOp Term term;

syntax  Term =   factorTerm:  Factor factor
    | multTerm:  Term term MultOp multOp Factor factor;

syntax  Factor = factor:  Ident name
    | factorInt:  Int intVal
    | factorFloat:  Float floatVal
    | factorColumn :  Ident name "."  Ident name
    | factorString:  String str
    | factorDate:  DateFunct dateFunct
    | factorExpr:  "("  ExprSimple exprSimple ")"
    | funcFactor:  Function function FuncParen funcParen
    | groupFactor:  GroupFunc groupFunction GroupFuncParen groupFuncParen;

syntax  Function = upper:  "upper"
    | lower:  "lower"
    | abs:  "abs"
    | len:  "length" ;

syntax  FuncParen = funcParenExpr:  "("  ExprSimple exprSimple ")"
    | funcParenParenDbl:  "("  FuncParenDbl funcParenDbl ")" ;

syntax  FuncParenDbl = funcParenDbl:  ExprSimple exprSimple1 ","  ExprSimple exprSimple2;

syntax  GroupFunc = avg:  "avg"
    | count:  "count"
    | max:  "max"
    | min:  "min"
    | sum:  "sum" ;

syntax  GroupFuncParen = groupExprSimple:  "("  ExprSimple exprSimple ")"
    | groupStar:  "("  "*"  ")" ;

syntax  WhereClause = where:  "where"  Condition condition;

syntax  Condition = condition:  LogicTerm logicTerm
```

```
    | bracketCondition:  "("  Condition condition ")"
    | notCondition:  "not"  LogicTerm logicTerm
    | orCondition:  Condition condition "or"  LogicTerm logicTerm
    | andCondition:  Condition condition "and"  Condition condition;

syntax  LogicTerm = logicTerm:  LogicFactor logicFactor
    | andTerm:  LogicTerm logicTerm "and"  LogicFactor logicFactor
    | bracketLogicTerm:  "("  LogicTerm logicTerm ")" ;

syntax  LogicFactor = comparison:  Comparison comparison
    | inclusion:  ExprSimple exprSimple "in"  Subquery subquery;

syntax  Comparison = simple:  ExprSimple exprLeft CompareOp compareOp ExprSimple exprRight
    | multiple:  Comparison comparison CompareOp compareOp ExprSimple exprRight
    | isNull:  ExprSimple expr "is"  "null" ;

syntax  AdditionalClauses = limitClause:  Limit limit
    | orderClause:  OrderBy orderBy
    | orderAndLimit:  OrderBy orderBy Limit limit;

syntax  Limit = limit:  "limit"  Int offset
    | limitWithRange:  "limit"  Int from ","  Int to;

syntax  OrderBy = orderByCol:  "order"  "by"  ExprSimple exprSimple
    | orderByColWithDirection:  "order"  "by"  ExprSimple exprSimple OrderDirection direction;

syntax  OrderDirection = asc:  "asc"  | desc:  "desc" ;

syntax  AddOp = add:  "+"  | sub:  "-" ;

syntax  MultOp = mult:  "∗"  | div:  "/" ;

syntax  CompareOp = gt:  "\>;"  | lt:  "\<;"  | eq:  "="  | ge:  "\>;="  | le:  "\<;="  |
ne:  "\<;\>;" ;

lexical  Ident = ([a - z  0 - 9  _ ] !<;<; [a - z ][a - z  0 - 9  _ ]∗ !>;>; [a - z  0 -
9  _ ]) | "?" ;

lexical  Int = [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  Float = [0 - 9 ]∗ "."  [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  String = "\""  StringChar∗ [\\ ] !<;<; "\""
     | "\'"  StringChar∗ [\\ ] !<;<; "\'" ;

lexical  StringChar = ![\" ] | [\\ ] <;<; [\" ];

lexical  DateFunct = currdate:  "curdate()"
     | now:  "now()" ;
```

Insert.rsc

```
module  lang::php::query::\syntax::Insert

layout  Standard = [\t \n \  \r \f ]∗;
```

```
start syntax Insert =
    \insert: "insert" "into" Table table "values" "(" FactorList values ")"
    | insertCols: "insert" "into" Table table "(" ColumnList colList ")" "values" "("
FactorList values ")" ;

syntax ColumnList = columns: {Column "," }+ columns;
syntax FactorList = values: {Factor "," }+ values;

syntax WhereClause = where: "where" Condition condition;

syntax Condition = condition: LogicTerm logicTerm
    | bracketCondition: "(" Condition condition ")"
    | notCondition: "not" LogicTerm logicTerm
    | orCondition: Condition condition "or" LogicTerm logicTerm
    | andCondition: Condition condition "and" Condition condition;

syntax LogicTerm = logicTerm: LogicFactor logicFactor
    | andTerm: LogicTerm logicTerm "and" LogicFactor logicFactor
    | bracketLogicTerm: "(" LogicTerm logicTerm ")" ;

syntax LogicFactor = comparison: Comparison comparison;

syntax Comparison = simple: ExprSimple exprLeft CompareOp compareOp ExprSimple exprRight
    | multiple: Comparison comparison CompareOp compareOp ExprSimple exprRight
    | isNull: ExprSimple expr "is" "null" ;

syntax Factor = factorColumn : Column column
    | factorInt: Int intVal
    | factorFloat: Float floatVal
    | factorString: String str
    | factorDate: DateFunct dateFunct;

syntax Term = factorTerm: Factor factor
    | multTerm: Term term MultOp multOp Factor factor;

syntax ExprSimple = addExpr: ExprSimple exprSimple AddOp addOp Term term
    | termExpr: Term term
    | unaryExpr: AddOp addOp Term term;

syntax Function = upper: "upper"
    | lower: "lower"
    | abs: "abs"
    | len: "length" ;

syntax FuncParen = funcParenExpr: "(" ExprSimple exprSimple ")"
    | funcParenParenDbl: "(" FuncParenDbl funcParenDbl ")" ;

syntax FuncParenDbl = funcParenDbl: ExprSimple exprSimple1 "," ExprSimple exprSimple2;

syntax AdditionalClauses = limitClause: Limit limit
    | orderClause: OrderBy orderBy
    | orderAndLimit: OrderBy orderBy Limit limit;

syntax Limit = limit: "limit" Int offset;

syntax OrderBy = orderByCol: "order" "by" ExprSimple expr
```

```
    | orderByColWithDirection:  "order"  "by"  ExprSimple expr OrderDirection direction;

syntax  OrderDirection = asc:  "asc"  | desc:  "desc" ;

syntax  Table = table:  Ident name
    | qtable:  "‘"  Ident name "‘" ;

syntax  Column = column:  Ident name
    | qcolumn:  "‘"  Ident name "‘" ;

syntax  AddOp = add:  "+"  | sub:  "-" ;

syntax  MultOp = mult:  "*"  | div:  "/" ;

syntax  CompareOp = gt:  "\>;"  | lt:  "\<;"  | eq:  "="  | ge:  "\>;="  | le:  "\<;="  |
ne:  "\<;\>;" ;

lexical  Int = [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  Ident = ([a - z  A - Z  0 - 9  _ ] !<;<; [a - z  A - Z ][a - z  A - Z  0 - 9  _
]* !>;>; [a - z  A - Z  0 - 9  _ ]) | "?" ;

lexical  Float = [0 - 9 ]* "."  [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  String = "\""  StringChar* [\\ ] !<;<; "\""
    | "\’"  StringChar* [\\ ] !<;<; "\’" ;

lexical  StringChar = ![\" ] | [\\ ] <;<; [\" ];

lexical  DateFunct = currdate:  "curdate()"
    | now:  "now()" ;
```

Update.rsc

```
module  lang::php::query::\syntax::Update

layout  Standard = [\t \n \  \r \f ]*;

start  syntax  Update = SingleTableUpdate;

syntax  SingleTableUpdate =
     update:  "update"  Table table "set"  UpdateList updateList
     | update:  "update"  Table table "set"  UpdateList updateList WhereClause whereClause
     | update:  "update"  Table table "set"  UpdateList updateList AdditionalClauses additionalClauses
     | update:  "update"  Table table "set"  UpdateList updateList WhereClause whereClause
AdditionalClauses additionalClauses;

syntax  UpdateList = updateList:  {Assign "," }+ assigns;

syntax  Assign = assign:  Column column "="  Factor factor
     | assignDefault:  Column column "="  "default" ;

syntax  WhereClause = where:  "where"  Condition condition;

syntax  Condition = condition:  LogicTerm logicTerm
     | bracketCondition:  "("  Condition condition ")"
     | notCondition:  "not"  LogicTerm logicTerm
     | orCondition:  Condition condition "or"  LogicTerm logicTerm
     | andCondition:  Condition condition "and"  Condition condition;

syntax  LogicTerm = logicTerm:  LogicFactor logicFactor
     | andTerm:  LogicTerm logicTerm "and"  LogicFactor logicFactor
     | bracketLogicTerm:  "("  LogicTerm logicTerm ")" ;

syntax  LogicFactor = comparison:  Comparison comparison;

syntax  Comparison = simple:  ExprSimple exprLeft CompareOp compareOp ExprSimple exprRight
     | multiple:  Comparison comparison CompareOp compareOp ExprSimple exprRight
     | isNull:  ExprSimple expr "is"  "null" ;

syntax  Factor = factorColumn :  Column column
     | factorInt:  Int intVal
     | factorFloat:  Float floatVal
     | factorString:  String str
     | factorDate:  DateFunct dateFunct
     | factorExpr:  "("  ExprSimple exprSimple ")"
     | funcFactor:  Function function FuncParen funcParen;

syntax  Term =   factorTerm:  Factor factor
     | multTerm:  Term term MultOp multOp Factor factor;

syntax  ExprSimple = addExpr:  ExprSimple exprSimple AddOp addOp Term term
     | termExpr:  Term term
     | unaryExpr:  AddOp addOp Term term;

syntax  Function = upper:  "upper"
     | lower:  "lower"
     | abs:  "abs"
     | len:  "length" ;

syntax  FuncParen = funcParenExpr:  "("  ExprSimple exprSimple ")"
    | funcParenParenDbl:  "("  FuncParenDbl funcParenDbl ")" ;

syntax  FuncParenDbl = funcParenDbl:  ExprSimple exprSimple1 ","  ExprSimple exprSimple2;

syntax  AdditionalClauses = limitClause:  Limit limit
    | orderClause:  OrderBy orderBy
    | orderAndLimit:  OrderBy orderBy Limit limit;

syntax  Limit = limit:  "limit"  Int offset;

syntax  OrderBy = orderByCol:  "order"  "by"  ExprSimple expr
    | orderByColWithDirection:  "order"  "by"  ExprSimple expr OrderDirection direction;

syntax  OrderDirection = asc:  "asc"  | desc:  "desc" ;

syntax  Table = table:  Ident name
    | qtable:  "‘"  Ident name "‘" ;

syntax  Column = column:  Ident name
    | qcolumn:  "‘"  Ident name "‘"
    | tableColumn:  Ident tableName "."  Ident colName
    | qtableColumn:  "‘"  Ident tableName "."  Ident colName "‘" ;

syntax  AddOp = add:  "+"  | sub:  "-" ;

syntax  MultOp = mult:  "*"  | div:  "/" ;

syntax  CompareOp = gt:  "\>;"  | lt:  "\<;"  | eq:  "="  | ge:  "\>;="  | le:  "\<;="  |
ne:  "\<;\>;" ;

lexical  Int = [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  Ident = ([a - z  A - Z  0 - 9  _ ] !<;<; [a - z  A - Z ][a - z  A - Z  0 - 9  _
]* !>;>; [a - z  A - Z  0 - 9  _ ]) | "?" ;

lexical  Float = [0 - 9 ]* "."  [0 - 9 ]+ !>;>; [0 - 9 ];

lexical  String = "\""  StringChar* [\\ ] !<;<; "\""
    | "\'"  StringChar* [\\ ] !<;<; "\'" ;

lexical  StringChar = ![\" ] | [\\ ] <;<; [\" ];

lexical  DateFunct = currdate:  "curdate()"
    | now:  "now()" ;
```

ParseSelect.rsc

```
module  lang::php::query::util::ParseSelect

import  ParseTree;
import  lang::php::query::\syntax::Select;

public  start [Select] parseSelect(str  src) = parse(#start [Select],
src);
```

ParseInsert.rsc

```
module  lang::php::query::util::ParseInsert

import  ParseTree;
import  lang::php::query::\syntax::Insert;

public  start [Insert] parseInsert(str  src) = parse(#start [Insert],
src);
```

ParseUpdate.rsc

```
module  lang::php::query::util::ParseUpdate

import  ParseTree;
import  lang::php::query::\syntax::Update;

public  start [Update] parseUpdate(str  src) = parse(#start [Update],
src);
```

ParseDelete.rsc

```
module  lang::php::query::util::ParseDelete

import  ParseTree;
import  lang::php::query::\syntax::Delete;

public  start [Delete] parseDelete(str  src) = parse(#start [Delete],
src);
```

Parse.rsc

```
module  lang::php::query::util::Parse

import  lang::php::query::util::ParseSelect;
import  lang::php::query::util::ParseUpdate;
import  lang::php::query::util::ParseDelete;
import  lang::php::query::util::ParseInsert;
import  String;
import  IO;

public  void  parse(str  queryString) throws  ParseError {
    str  lowerCaseQueryString = toLowerCase(queryString);
    if  (startsWith(lowerCaseQueryString, "select" )) {
        println("parsing select:  <; queryString>;" );
        parseSelect(lowerCaseQueryString);
    }
    if  (startsWith(lowerCaseQueryString, "update" )) {
        println("parsing update:  <; queryString>;" );
        parseUpdate(lowerCaseQueryString);
    }
    if  (startsWith(lowerCaseQueryString, "delete" )) {
        println("parsing delete:  <; queryString>;" );
        parseDelete(lowerCaseQueryString);
    }
    if  (startsWith(lowerCaseQueryString, "insert" )) {
        println("parsing insert:  <; queryString>;" );
        parseInsert(lowerCaseQueryString);
    }
}
```