Are PHP Applications Ready for Hack?

Laleh Eshkevari*, Fabien Dos Santos†, James R. Cordy‡, and Giuliano Antoniol*
*Department of Génie Informatique et Génie Logiciel, Ecole Polytechnique de Montréal, Canada
laleh.mousavi-eshkevari@polymtl.ca, antoniol@ieee.org
†Departement Informatique et Gestion, Ecole Polytechnique Universitaire de Montpellier, France
Fabien.dos-santos@polytech.univ-montp2.fr
‡School of Computing, Queen's University, Canada
cordy@cs.queensu.ca

***Abstract*—PHP is by far the most popular WEB scripting language, accounting for more than 80% of existing websites. PHP is dynamically typed, which means that variables take on the type of the objects that they are assigned, and may change type as execution proceeds. While some type changes are likely not harmful, others involving function calls and global variables may be more difficult to understand and the source of many bugs. Hack, a new PHP variant endorsed by Facebook, attempts to address this problem by adding static typing to PHP variables, which limits them to a single consistent type throughout execution.**

This paper defines an empirical taxonomy of PHP type changes along three dimensions: the complexity or burden imposed to understand the type change; whether or not the change is potentially harmful; and the actual types changed. We apply static and dynamic analyses to three widely used WEB applications coded in PHP (WordPress, Drupal and phpBB) to investigate (1) to what extent developers really use dynamic typing, (2) what kinds of type changes are actually encountered; and (3) how difficult it might be to refactor the code to avoid type changes, and thus meet the constraints of Hack's static typing.

We report evidence that dynamic typing is actually a relatively uncommon practice in production PHP programs, and that most dynamic type changes are simple representational changes, such as between strings and integers. We observe that most PHP type changes in these programs are relatively simple, and that the largest proportion of them are easy to refactor to consistent static typing using simple local renaming transformations. Overall, the paper casts doubt on the usefulness of dynamic typing in PHP, and indicates that for many production applications, conversion to Hack's static typing may not be very difficult.

***Keywords*—PHP, Dynamic typing, Type safety**

I. Introduction

Php is a scripting language, dynamically typed with no static type checking or variable declarations, and is supported by an ecosystem of hundreds of pre-defined, ready to use libraries, modules and functions. Development in such a rich ecosystem is at a faster pace; there is no need to compile, no need for complicated build files, and PHP code can be interwoven directly with HTML code making it very easy to develop custom websites and web development frameworks.

Avoiding the limitations of type safety is designed to increase flexibility and reuse - for example, we can concatenate strings, integers or mixes of them using the same dot operator, or we can assign an array to a variable that previously contained a boolean value. During execution or on different execution paths, a PHP variable may be assigned values of many different types, for example a variable can first be assigned an integer, and then a string or perhaps later an array.

For such type-changing assignments, in this paper we refer to the type of a variable before it is changed as "type-before", and the type after modification as "type-after". For example, in the following code fragment we see that an array is assigned to the variable `$contact` through the invocation of the function `str_split`. The following statement assigns a string value (using the `dot` operator) to the same variable.

```
$concat = str_split($concat,128);
$concat = 'load%5B%5D='. implode('
                    &load%5B%5D=',$concat);
```

However, such powerful language features also have a downside, and often come at the cost of late error identification, particularly in large codebases [5] PHP is no exception in this respect.

The benefits of static type checking have been well discussed in literature [25], [27], [34]. A recent empirical study by Ray *et al.* evaluated the impact of programming language choice on the quality of software. The results of the study show that the quality of systems developed in statically type languages is significantly better than those in dynamically type languages. Though other studies show the negative impact of static type languages on development time and productivity [30], [18]. Despite the discussed benefits and empirical supports for statically type languages, popularity of dynamic languages are increasing [26], [6]. Meijer *et al.* elaborate the need for a language that provides both safety and flexibility [23]. The desirability of having both a fast development pace and the capability of early error detection recently motivated the Facebook staff to develop the Hack programming language [1].

Hack can be viewed as PHP with the addition of static typing, thus ensuring type safety. PHP code in the Facebook codebase is converted to Hack using a set of custom code modification tools [2]. However, the automated code conversion may not have good coverage if the project uses certain dynamic features, such as `global`, `eval`, *Variable* variables, and other aspects of PHP that are not recognized in Hack. While one may expect that due to its static typing Hack programs may have better quality, fewer bugs and easier maintenance than PHP programs, to the best of our knowledge there is as yet no experimental evidence that this is the case. Thus, many PHP application teams may not yet be convinced to convert their programs to Hack, fearing that the overhead of the conversion effort may not be worth it.

In light of this, we are interested in understanding to what extent dynamic typing is actually used in PHP programs, and

978-1-4799-8469-5/15/$31.00 © 2015 IEEE

for what purposes. Our intuition is that dynamic typing makes a program more difficult to understand, and thus we suspect that in production PHP programs, very few variables may actually change type during execution. If this is indeed the case, then the effort to convert a PHP application to Hack by eliminating dynamic typing using source code refactoring may not actually be very high. Our goal is to provide a practical approach for identifying, classifying and reporting violations of type safety in a PHP application, and, armed with this knowledge, to leave the decision to the PHP developers on what action should be taken.

We perform an automated hybrid static and dynamic analysis of dynamic typing, complemented by manual refactoring and validation.We rely on dynamic analysis to identify type changes at run time using program instrumentation. We perform a static analysis to identify the scope of the variables, and finally we manually verify whether we can ensure type safety through renaming by refactoring and repeating the dynamic analysis. We apply the technique to four production open source PHP applications: phpBB2, phpBB3, Drupal, and Wordpress. All examples discussed are taken from systems we analyzed. The main contributions of this paper lie in the empirical taxonomy of type changes, the approach to automatically detect and classify type changes, as well as in the empirical validation on four production PHP applications.

Paper structure. Section II begins with a brief description of types in PHP and our proposed classification of type changes. Section III describes our approach to identifying and classifying type changes in PHP assignment statements using a combination of static and dynamic analysis of PHP code. Section IV reports on our empirical study aimed at demonstrating the approach's feasibility and analyzing the results with respect to our three research questions on four production PHP systems. Following a discussion of related work in Section VI, Section VII concludes the paper and outlines directions for future work.

II. Background

The PHP language has a number of standard types, for example strings (String), integers (Integer), floating-point numbers (Double), and arrays of these. PHP is dynamically typed, and thus it does not associate variables strictly with any specific type. As a result, executed assignments to a variable can cause it to take on multiple different types during the execution of a program. Type conversion in PHP can be explicit (casting) or implicit (coercing). The PHP manual [3] uses the term *type juggling* for the implicit conversion of types. Type juggling is automatically done when variables of different types are used as operands of a mathematical or logical operation. For example, in the code shown below, the variable `$foo` is a `boolean` (through explicit casting). Variable `$inc` is a `String` since a string value is assigned to it. However, variable `$inc` is implicitly coerced to be an `Integer` with value one, thus making `$val` an `Integer`.

```
$foo = (boolean) $bar;
$inc= "1";
$val= 2 + $inc;
```

We refer to a statement that changes the type of a variable as a "type changing statement". All assignment statements in PHP are potentially type changing, changing the type of the variable on the left hand side to the type of the value on the right. In this study we consider only variables and the type changes due to assignments to them. Temporary type changes, such as the implicit coercion of the value of variable `$inc` from `String` to `Integer` in the example above, are ignored. However, in the following code, the last assignment will be marked as "type changing statement" that changes the type of `$inc` from `String` to an `Array`, since the change persists after the assignment.

```
$foo = (boolean) $bar;
$inc= "1";
$val= 2 + $inc;
$inc= array ();
```

We classify type changing statements across three dimensions: (1) complexity (trivial versus non-trivial type changes), (2) non-harmful versus potentially-harmful, and (3) according the actual type change made, from type-before to type-after.

A. Trivial and Non-trivial Type Changes

Here the term "trivial" is meant to represent the effort likely needed for a non-expert on the code to understand the change between the relationship between the type-before and the type-after of an assigned variable. Thus some type changes are trivial, that is, by looking at the right hand side (RHS) of the type changing statement we can know the type of left hand (LHS) side without any other information. For example, in this snippet from phpBB3, the variable `$root_data` is changed from `String` to `Array` due to an explicit call to the `array` constructor.

```
$root_data = array('forum_id' => 0);
```

Changes in type can also be due to a simple change in data representation. For example, in this code from phpBB3, the `String` value of variable `$user_permissions` is changed to `Array` by calling the function `explode`, which splits a given string value into an array based on a given delimiter.

```
$user_permissions = explode("\n",
                            $user_permissions);
```

The documentation for the function/method being called on the RHS helps, if it exists and is up-to-date, to simplify the task of understanding the after-type of the variable assigned. Unfortunately, there are some type changes that are neither explicit nor due to simple changes in data representation. For example, in the following code fragment from Drupal, the variable `$display` is changed from `String` to `Array`, with no local indication of the type.

```
$display = field_get_display($instance,
                                  $view_mode,$entity);
```

In this example, when analyzing the type of the RHS, neither the documentation nor the implementation of the function is helpful in identifying the type-after. For such cases, only the developer of $field\_get\_display$ or someone very familiar with the code will be able to guess the type-after.

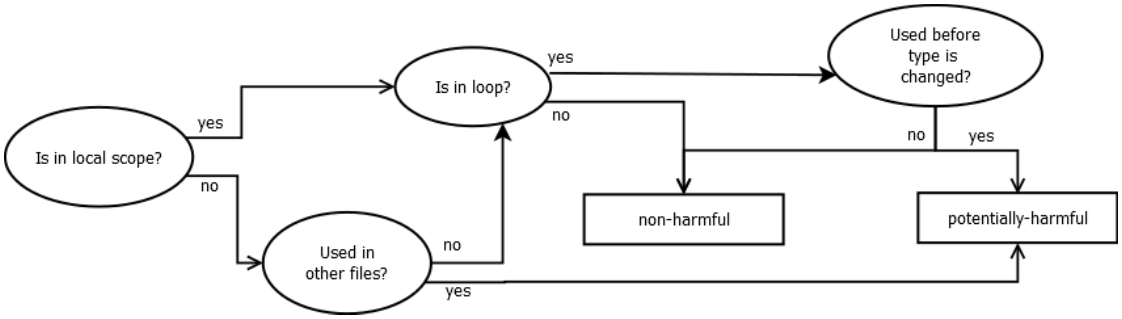


Fig. 1. Potentially-harmful vs. non-harmful labelling of type-changing assignments.

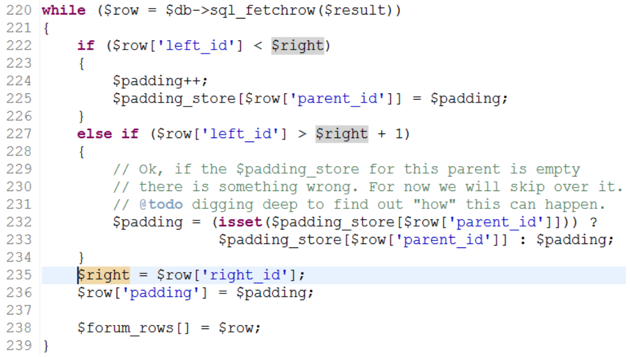
In a nutshell, this classification is based on the likely difficulty or effort required for a new programmer to infer the after-type of the variable from the code.

B. Non-harmful and Potentially-harmful Type Changes

One simple way to refactor a type change to make it type safe is to simply rename the variable on the LHS of the type changing statement. However, such a renaming requires that all uses of the variable are consistently replaced with the new name[1]. Unfortunately, such renaming is not always straightforward. For example, one cannot simply rely on the identity of variable names in different files; different files may or may not intend to refer to the same variable. Understanding the relationships between variable uses and definitions in PHP requires that all file include relations are resolved, which may not be possible without running the program, and even when possible, doing it by hand can be a daunting task.

We have defined criteria to automatically classify type changes into non-harmful and potentially-harmful based on the scope of the type changing statements and their reaching definition information. To this aim we have implemented the flow-sensitive and context-insensitive reaching definition algorithm proposed by Tonella *et al.* [32]. The idea underlying the classification is based on how difficult it might be to remove the type change using a local renaming of the LHS variable. The idea is that non-harmful type changes can be locally renamed to eliminate type changes easily by introducing a new variable as the target of the assignment, while renaming potentially-harmful ones would need to be done very carefully and with a more global analysis of variable references. Figure 1 illustrates the logic behind the decision process for assigning a type changing statement to one of these two categories.

Predicate “is in local scope?": The predicate “is in local scope?" is true if the variable is local to a function/method. Thus, if the variable is declared to be global inside a function/method through the use of keyword `global`, the predicate will evaluate to false. PHP functions may also have “static" variables, which do not lose their value when execution leaves the function. In such cases the predicate will also evaluate to false, if the type changing statement is the last statement in the function to modify the variable, which means that the new value (and hence the new type) of the variable will be visible to the next execution of the function.



```
220 while ($row = $db->sql_fetchrow($result))
221 {
222     if ($row['left_id'] < $right)
223     {
224         $padding++;
225         $padding_store[$row['parent_id']] = $padding;
226     }
227     else if ($row['left_id'] > $right + 1)
228     {
229         // Ok, if the $padding_store for this parent is empty
230         // there is something wrong. For now we will skip over it.
231         // @todo digging deep to find out "how" this can happen.
232         $padding = (isset($padding_store[$row['parent_id']])) ?
233                 $padding_store[$row['parent_id']] : $padding;
234     }
235     $right = $row['right_id'];
236     $row['padding'] = $padding;
237 
238     $forum_rows[] = $row;
239 }
```

Fig. 2. The type of $right changes from Integer to String on line 235.

Predicate “used before type is changed?": To evaluate the result of this predicate, we rely on the reaching-definition analysis. The predicate is true if the type changing statement is inside a loop and there exists a statement in loop that satisfies the following two conditions:

- The statement appears before the type changing statement.
- The statement uses the variable that is the LHS of the type changing statement.

For example, in phpBB3 the type of variable `$right` is changed from `Integer` to `String` on line 235 (Figure2). Variable `$right` on LHS of line 235 reaches to line 222 and 227. Thus the type change statement on line 235 is classified as potentially-harmful.

Predicate “used in other files?": Variables that are defined in a PHP file can be used in other PHP files through the PHP file inclusion mechanism. The reaching definition analysis enables us to identify all statements that are possibly using a variable whose type is changed. It must be noted that, at this stage, we have not implemented a points-to or reference analysis - thus our results are approximated: there may be cases where a variable is passed by reference to a function and changed inside the function. In such a case we assume that it is actually changed at the call site. This predicate evaluates to true if the variable on the LHS of a type change statement is used in a statement in another PHP file.

C. Identifying Type-changing Statements

To find all type-changing statements, we instrument all variable assignments in the application under study, logging the

[1] To the best of our knowledge, open source editors of PHP do not support automatic renaming.

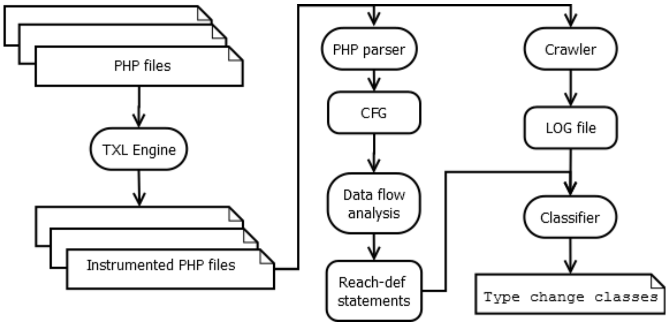


Fig. 3. Type change identification and classification process.

type-before and the type-after for each executed assignment in which the type changes. Of course, just because a type does not change in the coverage tests running our dynamic analysis, we cannot exclude the possibility that in some other execution scenario it might change - this is a limitation intrinsic to dynamic analysis. However, by covering all pages and links in the application, we attempt to mitigate this possibility.

III. Proposed Approach and Tool Support

Our approach to identifying and classifying type changes uses a combination of static and dynamic analysis (Figure 3).

A. Dynamic Analysis

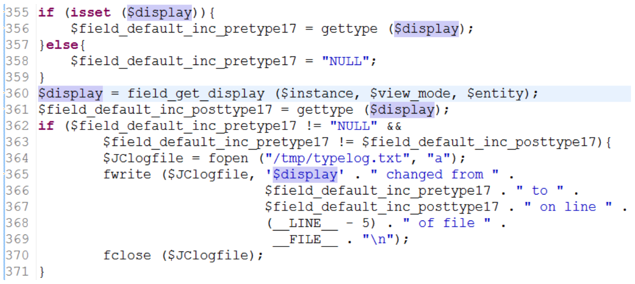
We use TXL [12] source transformation to add run-time instrumentation to the PHP code of the systems under analysis to detect and log the type-before and type-after of each type-changing assignment. Type change is detected by inserting code to query the type of the LHS variable before and after the assignment using the PHP `gettype` function, and comparing the type-before to the type-after. If the two differ, then a log file entry is made for the change. By executing these checks as part of the actual execution of the WEB application, we obtain exact type change information.

Figure 4 shows an example code fragment from Drupal after the instrumenting transformation. The TXL transformation inserts statements before (lines 355-359) and after (lines 361-371) each PHP assignment statement (line 360) to obtain the variable's type using the `gettype` PHP function. The string value “NULL" is used to mark the case where the variable was not previously assigned (line 358), so that first assignments are not logged as type changes.

If the types of the variable before and after the assignment are different (line 362-363) then we enter the change into the type change log file with the name of the variable, the before-type, the after-type, and the source line and file coordinates of the assignment statement (lines 364-370).

B. Crawling the WEB Application

We use the Ruby-based open source tool Watir [4] to mimic the way that a user interacts with the WEB site implemented by the WEB application. Watir can perform all of the same actions that a user does: it can open links, fill in forms, press buttons, and so on. To thoroughly exercise and cover execution of the



```
355 if (isset ($display)){
356     $field_default_inc_pretype17 = gettype ($display);
357 }else{
358     $field_default_inc_pretype17 = "NULL";
359 }
360 $display = field_get_display ($instance, $view_mode, $entity);
361 $field_default_inc_posttype17 = gettype ($display);
362 if ($field_default_inc_pretype17 != "NULL" &&
363         $field_default_inc_pretype17 != $field_default_inc_posttype17){
364         $JClogfile = fopen ("/tmp/typelog.txt", "a");
365         fwrite ($JClogfile, '$display' . " changed from " .
366                             $field_default_inc_pretype17 . " to " .
367                             $field_default_inc_posttype17 . " on line " .
368                             (__LINE__ - 5) . " of file " .
369                              __FILE__ . "\n");
370         fclose ($JClogfile);
371 }
```

Fig. 4. Code fragment instrumentation.

PHP code of each application, we implemented a custom Ruby “spider" script for the application. Each spider script customizes Watir to visit all of the pages, buttons, links and forms of the WEB application's default configuration, including at least one registered user and one forum, post or document entry of each kind supported by the application. Our spiders explore interactions in all of the various user roles of the application, such as administrator, registered user, and guest user. Coverage is measured by analyzing the Watir log to ensure that every PHP page of the application has been visited at least once.

C. Static Analysis

We use the Eclipse PDT parser to parse and build the control flow graph (CFG) of the PHP application under study. We first parse each PHP file and build a CFG graph for each file. To build the CFG of the entire system, we link the file level CFGs, resolving include relations and function/method calls. To resolve call bindings, we first use a name heuristic: if there exists only one declaration of a function/method of the name being called, then we bind the invocation to this instance. In cases where there are multiple functions/methods with the same name, we rely on the Eclipse PDT to resolve the binding.

In PHP, file inclusion is done using the `include` statement, which accepts an expression as argument. There are no constraints on the include expression, which can contain variables, calls to functions, and string operators. In other words, often the file name path is dynamically computed and built at run-time.

To avoid circular inclusions, PHP provides a number of different include statements, `include`, `include_once`, `require` and `require_once`. `include` and `require` always include the file passed as parameter. The difference is that `require` produces a compiler error upon failure. `include_once` and `require_once` work similarly to `include` and `require`, however do not include a file if it has been already included, avoiding a multiple inclusion. We use the approach proposed in our previous work [16] to resolve file inclusions in PHP.

After the CFG graphs are connected by call invocations and file inclusion, we perform the flow sensitive, context insensitive reaching definition analysis proposed by Tonela *et al.* [32] on the system-level CFG. We begin the propagation from the first node of the CFG graph, which normally corresponds to the home page of the application `index.php`, and traverse the CFG graph, iterating until the flow information ceases to change.

TABLE I. ANALYZED PHP SYSTEMS

System	Revision	PHP files	classes	functions/ methods	Total LOC
phpBB2	2.0.23	78	11	2,83	40,649
phpBB3	3.0.12	271	207	1,879	187,483
Drupal	7.26	275	99	3,322,	152,285
Wordpress	3.8	483	243	8,358	228,748

Using a combination of the logged type changes (dynamic analysis), scope information for the type changed variables (from the program's Eclipse AST), and the flow propagation information, we have enough information to classify the type changes according to the categories of Section II.

IV. Empirical Study

The *goal* of this study is to use the approach described in Section III, with the *purpose* of identifying type changes in PHP application, and evaluating the effort needed to ensure type safety. The *context* of the study consists of four PHP open source applications: phpBB2-2.0.23, phpBB3-3.0.12, Drupal-7.26, and Wordpres-3.8 (Table I).

A. Research Questions

The goal of this study is to address the following research questions:

- **RQ1**: *How frequently are variable types changed?* The aim of this research question is first to assess what variables' types change in the PHP application, and how frequently.
- **RQ2**: *What kinds of type changes are used?* The goal of this research question is to classify the observed type changes in the three dimensions outlined in SectionII.
    - **RQ 2.1** *Are these type changes trivial?* As explained in Section II, in many cases the type-after of a type change is directly evident from the right hand side (RHS) of the assignment, making it easier to understand. For example, use of the `dot` operator in the main expression of the RHS indicates that the new type of the LHS variable will be `String`. The source code or the documentation of a function/method being invoked on the RHS may enable us to identify the type-after of the LHS as well. We refer to assignment statements in which the type-after is identifiable only by evaluating the RHS as "trivial" type change statement. The purpose of this question is to evaluate the code comprehension overhead of the type change from the point of view of a new programmer or maintainer of the code. We conjecture that many type changes in PHP are of this kind, and perform a qualitative analysis to answer to this research question.
    - **RQ 2.2** *Are these type changes harmful?* Based on the decision process outlined in Section II (Figure 1), we classify the observed type changes as either non-harmful and potentially-harmful. The classification is based on the scope of the variable on LHS of the assignment statements and its uses. This research question addresses the type changes from a risk of error point of view.
    - **RQ 2.3** *What are the type changes?* The goal of this research question is to examine the observed set of type changes in system under analysis in detail, and compile them into a catalogue of before-after type pairs for further analysis.
- **RQ3**: *Can we ensure type safety?* The goal of this research question is to investigate the effort needed to ensure type safety. We would like to know what proportion of the observed type changes can be avoided through simple renaming or lightweight refactoring of the code. We manually apply renaming/refactoring where possible, document the effort needed, and validate the changes by retesting the application. Our conjecture is that non-harmful type changes require much less effort in renaming/refactoring to avoid type changes.

TABLE II. PAGE COVERAGE.

System	Total pages in default config	Total pages covered	Percentage of page coverage
phpBB2	30	30	100%
phpBB3	13	13	100%
Drupal	4	4	100%
Wordpress	39	39	100%

B. Case Studies

This section reports quantitative and qualitative results of the analysis of four production WEB applications with respect to the three research questions formulated above. All type changes, and the classification are available for download on-line[2].

Type change frequency: As it was explained in Section III, for each of the four systems under study a Watir spider script was created to simulate a WEB application session exploring the WEB site in each user role, following all links and pressing all buttons in the default configuration of the system with one registered user and one forum, blog or document. In order to evaluate coverage, we logged visits to PHP pages as part of the spider. While we cannot insure that all of the PHP code in the application is exercised by the spider script, we can be sure that all PHP pages of the application have been visited. Table II shows number of PHP pages visited for each system, as well as the percentage of page coverage (100% in all cases).

In order to estimate the proportion of assignment statements that are type changing, we need to count the number of assignment statements in the WEB application source, and the number of those that are observed to be type-changing at run time when the application is thoroughly exercised. Table III shows these statistics for the four systems under study. The first row in the table is the total number of assignment statements in the WEB applications PHP code. The second row is the number of type-changing assignments executed during the spider's exercising of the entire web site, and the third row is the unique number of assignment statements involved in these changes. The percentage of assignment statements observed to be type-changing is shown in the bottom row, row 3 divided by row 1.

It is important to note that while we can be sure of the run-time coverage of all application pages, links and buttons, and thus the code that implements them, due to data gathering limitations we cannot be certain that all assignments have been executed in our present implementation. Thus our percentages

[2] http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/php-type-changes.tar.gz

TABLE III. RQ1: PROPORTION OF TYPE-CHANGING ASSIGNMENTS IN ANALYZED SYSTEMS

	phpBB2	phpBB3	Drupal	Wordpress
Total assignment stmts	4,830	13,655	7,261	18,982
Total run-time type changes	287	19,375	43,8074	263,617
Unique type changing stmts	35	103	63	177
Percentage	0.72%	0.80%	0.95%	0.92%

TABLE IV. RQ 2.1: TRIVIAL VS. NON-TRIVIAL TYPE CHANGES

	phpBB2	phpBB3	Drupal	Wordpress
Trivial	20	48	37	132
Non-trivial	15	55	26	45
Total	35	103	63	177

TABLE V. RQ 2.1: LOCAL VS. GLOBAL NON-TRIVIAL TYPE CHANGES

System	Non-trivial	Local & non-trivial	Global & non-trivial
phpBB2	15	3	12
phpBB3	55	33	22
Drupal	26	23	3
Wordpress	45	41	4

TABLE VI. RQ 2.2: POTENTIALLY-HARMFUL VS. NON-HARMFUL TYPE CHANGES

	phpBB2	phpBB3	Drupal	Wordpress
Non-harmful	14	79	49	165
Potentially-harmful	21	24	14	12

may be a slight under-estimate. However, a function coverage test insures us that all assignments were at least potentially executed by our Watir spider script.

Trivial and non-trivial type changes: Next we perform a qualitative analysis of the observed type changes to evaluate what proportion of them can be categorized as trivial. The observed number of actual type changing statements is relatively small (Table III), so we can perform this analysis by hand.

As explained in Section II, one can often determine the type-after of a variable in an assignment from the form or content of the RHS of the assignment. For example, if concatenate `dot` operator is the main expression operator in the RHS, then the type of variable will be `String`, or the source code or documentation of the main function being called in the RHS may tell us its type.

We manually examined the RHS of all assignment statements that were observed to be type changing during the execution of each program (Table III, row 3), to determine if the type-after was easily determined from the statement. Table IV shows the results of this analysis. With the exception of phpBB3, we can see that for more than half of the observed type-changing assignment statements, we can easily identify the type-after of the variables from the source code.

The results in this table become even more interesting when we take into account the scope of the variables (Table V). For example, phpBB3 has the largest proportion of non-trivial type changes (55 cases or 53%), but only 22 of such cases have global scope, and for both Drupal and Wordpress, almost all non-trivial type changes are to local variables. It is evident that if there is a possible threat due to type changes, the likely impact is more severe if the variable has global scope, so the fact that most non-trivial type changes are local rather than global is important.

Potentially-harmful and non-harmful type changes: As outlined in Section III, we first used our automatic classifier to classify the type changes as non-harmful (*i.e.*, locally scoped, and therefore easily renamed) or potentially-harmful (*i.e.*, requiring global analysis to rename), and then manually validated the results. Table VI shows the result of this classification.

The relatively small number of cases allowed us to perform a manual validation of every case, and thus to estimate the precision and the recall of the automated classifications (see Table VII and VIII). The manual validations were done using manual code tracing by one of the authors, using an independent second opinion in cases of doubt.

TABLE VII. RQ 2.2: PRECISION OF THE AUTOMATED ANALYSIS

	phpBB2	phpBB3	Drupal	Wordpress
Non-harmful	100%	95%	100%	99%
Potentially-harmful	67%	96%	78%	67%

TABLE VIII. RQ 2.2: RECALL OF THE AUTOMATED ANALYSIS

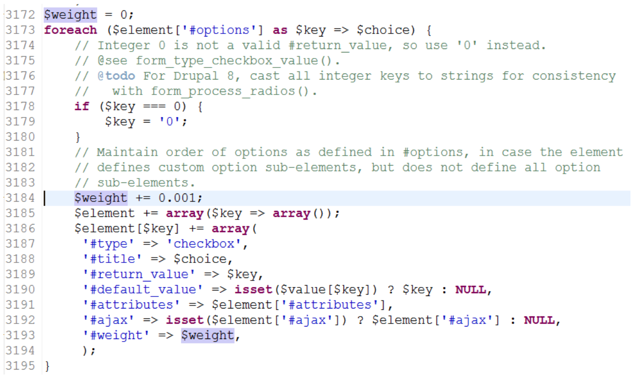
	phpBB2	phpBB3	Drupal	Wordpress
Non-harmful	67%	99%	94%	98%
Potentially-harmful	100%	85%	100%	89%

Unfortunately, sometimes theory and practice do not match: sometimes a type change that theoretically is potentially-harmful, when manually evaluated, is classified as non-harmful. For this reason our automated analysis did not yield 100% precision. For example, in the code fragment in Figure 5, the type of variable `$weight` is changed from `Integer` to `Double` at line 3184 in file `Drupal/includes/form.inc`. This type change is automatically classified as potentially-harmful by the flow analysis.

However, when we examine the code closely, we can see that variable `$weight` is used by two statements in the `foreach` loop: 3184, and 3186. The statement at line 3184 is the one that changes the type and also uses the variable. The statement at line 3186 always receives a double value for variable `$weight`, and thus there will never be a case where this statement uses `$weight` with any type other than double. Thus, we can see that this type change is non-harmful.

Another source of mis-classification by our automated analysis is the lack of reference and alias analysis in our process. If a reference to a variable is sent to a function/method, any change to the type of variable is visible outside of the scope of the function/method, and thus such type changes should be classified as potentially-harmful.

Our classification relies heavily on reaching-definition analysis. As explained in Section III, the propagation is done over the include graph of the system. In some cases, we cannot resolve the file being included since the path of file will only be known only on run time. This means that the include graph is not 100% complete. For those variables that are in the global scope, this lack of a precise include graph may lead to mis-classification. We therefore decided to use a conservative approach and classify such cases as potentially-harmful. Thus for some of the type changes that are automatically classified



```
$weight = 0;
foreach ($element['#options'] as $key => $choice) {
    // Integer 0 is not a valid #return_value, so use '0' instead.
    // @see form_type_checkbox_value().
    // @todo For Drupal 8, cast all integer keys to strings for consistency
    //   with form_process_radios().
    if ($key === 0) {
        $key = '0';
    }
    // Maintain order of options as defined in #options, in case the element
    // defines custom option sub-elements, but does not define all option
    // sub-elements.
    $weight += 0.001;
    $element += array($key => array());
    $element[$key] += array(
     '#type' => 'checkbox',
     '#title' => $choice,
     '#return_value' => $key,
     '#default_value' => isset($value[$key]) ? $key : NULL,
     '#attributes' => $element['#attributes'],
     '#ajax' => isset($element['#ajax']) ? $element['#ajax'] : NULL,
     '#weight' => $weight,
     );
}
```

Fig. 5. RQ 2.2: Example of a non-harmful type change that is mis-classified by the automated analysis.

as potentially-harmful, we manually verified that there is no way that such variables could reach to other files, and thus the type change is non-harmful. This effect leads to the lack of complete recall observed in Table VIII.

Detail of observed type changes: Table IX shows the result of our detailed analysis of the type changes identified in the analyzed systems. Entries in bold text indicate categories of type changes observed in all four systems, whereas another entries were observed in only some of the systems. It is important to note that following PHP language specification we did not differentiate arrays from dictionaries, stack, list, or other collections.

Overall, 46% of the type changes from array → string represent the joining of array elements with a given suffix, and 28% of the type changes from string → array are due to splitting the array content into strings based on a given delimiter. In PHP any non-zero integer can be interpreted as a `true` boolean value, and thus the change from boolean → integer (or vice versa) does not negatively impact any statement that uses a boolean (or integer, in the other case). We did not observe any other recurring patterns corresponding to the other type changes in the table. It is important to note that the total numbers in Tables IV and IX are different for phpBB3, Drupal and Wordpress. The reason is that a single assignment statement may be executed multiple times during a single execution of the application with different type-after results. For example, the assignment from Wordpress shown below was observed to change the type of variable `$res` from `boolean` to `array` on one occasion, and from `boolean` to `object` on another. Thus, the type changing statement is counted once as a type-changing statement, but twice as two different type changes.

```
$res = maybe_unserialize
    (wp_remote_retrieve_body ($request));
```

Ensuring type safety: As an experiment in transitioning to the type safety required by Hack, we attempted to manually modify all observed type-changing assignments using renaming and refactoring to avoid type changes, and documented our effort based on the Likert scale as easy, medium or hard. The renaming/refactoring was done by one of the authors, with a second author verifying the renaming/refactoring and the effort assigned. The resulting renamed/refactored code was re-run using the Watir spider script to evaluate and validate our changes.

TABLE IX. RQ 2.3: OBSERVED TYPE CHANGES IN THE ANALYZED SYSTEMS

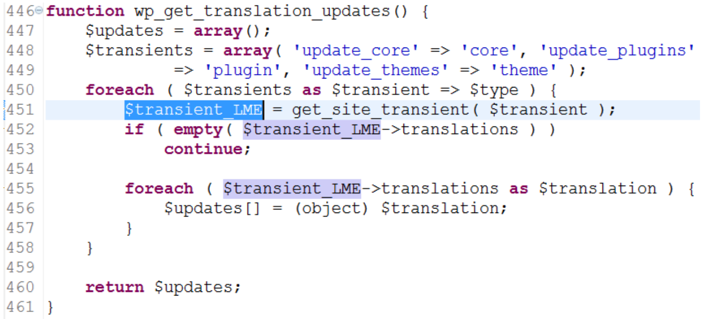
From→ to	phpBB2	phpBB3	Drupal	Wordpress
array→boolean	0	1	2	3
array →double	0	1	0	0
array →integer	0	0	1	4
array →NULL	0	0	6	0
array →object	0	0	1	6
array →string	5	9	8	21
boolean →array	3	9	5	24
boolean →integer	1	7	1	5
boolean →NULL	0	0	0	1
boolean →object	0	1	0	5
boolean →string	0	15	7	18
double →integer	0	0	1	1
double →string	0	0	1	1
integer →array	0	6	1	1
integer →boolean	2	3	0	0
integer →double	1	1	2	1
integer →NULL	0	0	1	1
integer →object	0	3	0	10
integer →string	9	22	6	13
object →array	0	1	1	4
object →boolean	0	2	0	0
object →integer	0	3	0	2
object →NULL	0	0	0	1
object →string	0	0	2	5
string →array	1	11	11	28
string →boolean	2	5	2	4
string →double	0	0	0	1
string →integer	11	8	3	17
string →NULL	0	1	5	5
string →object	0	0	0	11
Total	35	109	67	193

The goal of renaming/refactoring was not to strive for an elegant implementation, but instead to avoid the type change using the most minimal, simplest change to the code. Table X shows the result of renaming/refactoring for the systems. The second column refers to the number of cases for which we could avoid type change through renaming/refactoring.

TABLE X. RQ3: MANUAL REFACTORING TO AVOID TYPE CHANGES.

System	Can avoid	Cannot avoid	Uncertain
phpBB2	24	6	5
phpBB3	76	14	13
Drupal	46	17	0
Wordpress	167	8	2
Total	313	45	20

The results of this experiment were mixed. For phpBB2, we are able avoid type change by simple renaming/refactoring in 68% of the cases. whereas for Wordpress we are able to rename/refactor 94% of the cases. For both phpBB3 and Drupal, about 73% of the type changes can be avoid through simple renaming/refactoring. It is important to note that the uncertain cases are those for which we could proceed if we had complete knowledge about file inclusions. We also restricted ourselves to very local, simple code modifications in rode to reduce the risk of introducing new bugs into the system. We believe that a great many more cases could be easily avoided if the refactoring were done by the regular maintainers of the systems.



```
446 function wp_get_translation_updates() {
447     $updates = array();
448     $transients = array( 'update_core' => 'core', 'update_plugins'
449             => 'plugin', 'update_themes' => 'theme' );
450     foreach ( $transients as $transient => $type ) {
451         $transient_LME = get_site_transient( $transient );
452         if ( empty( $transient_LME->translations ) )
453             continue;
454
455         foreach ( $transient_LME->translations as $translation ) {
456             $updates[] = (object) $translation;
457         }
458     }
459
460     return $updates;
461 }
```

Fig. 6. RQ3: Example of loop-dependent type change.

Table XI shows the estimated effort required to avoid the type changes we were able to eliminate using renaming/refactoring. About 85% of the refactorings were estimated to be easy, and there is only one case which we labeled as hard. This is a case where the type change statement is located in a very long method (about 2,300 lines of code) where the entire logic is implemented as a sequence of nested switch and if statements. Thus the vast majority of changes seem to be easy. This result was expected, since majority of the type changed variables were observed to be local in scope and used in only a few statements.

TABLE XI. RQ3: EFFORT FOR MANUAL RENAMING/REFACTORING.

System	Can avoid	Easy	Medium	Hard
phpBB2	24	15	9	0
phpBB3	76	56	19	1
Drupal	46	40	6	0
Wordpress	167	155	12	0
Total	313	266	46	1

To evaluate whether our manual refactorings are effective, we re-instrumented the modified Wordpress code and once again exercised the modified application using the Watir spider script, crawling the modified WEB application as described in Section III to find any remaining type changes.

Of the 167 cases we were able to rename/refactor, the vast majority of the previously observed type changes had been resolved, and the statements were no longer reported as type-changing. However, in eight of the 167 cases, the refactored statements were still reported as type-changing. When we analyzed these cases, we observed that all of them were involved in loop structures like the one shown in Figure 6.

In this example, from Wordpress, the type of variable `$transient` is changed from `String` to `Object` on line 451 of file `/include/update.php`. The variable is initialized in each iteration of the for loop at line 450 and then the type-changed by the assignment on line 451. In the refactoring to resolve this type change, we had renamed the variable on the left hand side of line 451 to `$transient_LME`. Unfortunately, in the refactored code we observed that `$transient_LME` once changed from `Object` to `boolean`, and then from `boolean` back to `Object`. This is because in each iteration of loop the function `get_site_transient` returns different type. The author's comments on the refactoring indicated that there was some uncertainty about the effect of the loop on this change, and that indeed turned out to be the problem.

Summary: In summary, we can answer each of our research questions as follows:

- **RQ1:** For the four production PHP systems we analyzed, less than 1% of the assignment statements were observed to change types. It seems that despite the flexibility of dynamic typing in PHP, in practice production developers are relatively consistent in variable typing.
- **RQ2:** Overall we observed that more than half of the type change statements we found were trivial, that is, the result type of the assignment should be obvious to the programmer from the source code. Moreover, majority of the non-trivial type changes are local in scope and thus the impact of the type change is limited and relatively easy to resolve. The majority of the type changes we observed were judged to be non-harmful, and relatively simple changes from `string` → `array` and `integer` → `string` are the most frequent.
- **RQ3:** Our manual refactoring revealed that most non-harmful type changes can be avoided relatively easily using local renaming or refactoring. The result of re-running the refactored instrumented code of one large system showed that these simple renaming refactorings can be effective in the vast majority of cases.

V. Threats to Validity

Given the exploratory nature of the study, we mainly have threats to *construct* and *external* validity.

Construct validity threats concern the relationship between theory and observation. As we have explained in Section IV, one imprecision that could have occurred in our analyses is related to incomplete include graph. Thus we conservatively classified type changes of variable at global scope potentially-harmful in cases where such variables are used in other files. Moreover, we did not take into account the references passed to function/method while performing reaching-definition analysis. When we manually verified the result, we found that there were actually very few such cases in our subject systems.

While we validated execution coverage of the WEB applications at the page level, the lack of a statement-level measure of application coverage may affect our estimates of the actual proportion of assignments that are type-changing. Complete coverage of assignment statements in a WEB application is very difficult to insure, due to the large proportion of exception- and event-driven code. While we have mitigated this threat using a subsequent function coverage analysis, complete coverage of assignment statements can not been guaranteed, and thus our estimates may be slightly low.

External validity threats concern the generalizability of our results. The effort associated with the renaming/refactoring task is purely subjective *i.e.*, programming skill, and familiarity with the code can affect how one estimates the effort of refactoring. To limit the impact of this uncertainty, the refactorings and associated effort estimated by one of the authors was verified by a second author. A more realistic measure can be only provided by an experiment involving the actual developers of the systems.

We have no way to be certain that the renaming refactorings faithfully preserve the original behaviour. While we did not observe any visible change in behaviour when running the refactored code compared to the original, it is possible that more thorough testing might uncover some such changes.

Our study was limited to four of the most popular medium-to-large production open source PHP applications. As mentioned in SectionII, We only consider assignments to variables, thus type changes that are results of deserialization of data using JSON, or changes to the type of elements of arrays are not considered in this study, and thus our observation is limited to change of variables' type only. However, if a variables that holds data through JASON deserialization is reassigned we can detect if its type is changed. While we expect that we would see similar results for other PHP applications, more studies would need to be conducted to verify such a conjecture.

VI. Related Work

The focus of this paper is the use of dynamic and static analysis to explore the use of dynamic types in production PHP-based open source WEB applications. Relevant related work covers a number of areas: PHP analysis, WEB application reverse- and re-engineering, and type analysis for PHP and other languages.

A. PHP Application Analysis

Given its dynamic nature, PHP analysis often uses a mix of static and dynamic analysis. Nguyen *et al.* [24] developed a tool, WebDyn, for dynamic refactoring of PHP Web applications. They manually analyzed 2,664 revisions of four open-source PHP-based Web applications, and found that there exists an special form of refactoring that is specific to dynamic Web applications. Next, they categorized these refactorings (which they called output-oriented refactoring operations) in five groups: 1) dynamicalization (*e.g.*, replacing inline HTML/Javascript code with a PHP fragment or function), 2) re-structuring server and client code, 3) renaming embedded HTML/Javascript elements, 4) standardizing embedded HTML code, and 5) refactoring for separation of concerns. They use dynamic analysis coupled with symbolic execution to identify variable declarations, references as well as dangling references.

Alalfi *et al.* [7] proposed an approach based on a combination of static and dynamic information to analyze Web applications, and suggested the use of coverage metrics [8] to ensure accurate information. They applied this approach to the security analysis of WEB applications.

We share with these methods the use of a combination of static and dynamic information. Like Alalfi [8], we opted for a lightweight approach using a TXL source transformation to add instrumentation to the WEB application, complemented with a static analysis. Our goal however is very different, as we seek to quantify the use of dynamic typing and to develop a refactoring strategy to remove dynamic types by renaming and refactoring to static typing like that in Hack.

B. Web Application Reverse Engineering

While our overarching goal is clearly different, certain commonality can be found with the reverse engineering of WEB applications (WAs), in particular static and dynamic analysis. The first significant contribution was given by Ricca and Tonella, who developed the *ReWeb* tool to perform analyses on web sites [28], [29]. In particular, Ricca and Tonella introduced a graphical representation of the web site to allow for traditional static flow analyses such as reachability, dominance, and data flow analysis on WAs. The same authors also proposed to enhance static analysis by using dynamic information [31]. Clearly, *ReWeb* does not need page instrumentation; on the other hand, web server logs, do not allow fine-grained analyses such as needed to detect of a variable is being assigned a different types in two dfferent execution paths.

Di Lucca *et al.* [13], [14], [15] proposed an approach and a tool to extract Conallen's UML documentation, use cases and business object from Web applications. Their approach uses static analysis, however they pointed out that diagrams can be refined using dynamic information. *WARE* performs static analyses on WAs, stores the extracted information into a database and then uses such an information for the reverse-engineering of UML diagrams.

Architectural recovery was also the goal of the works of Hassan [19] and Antoniol *et al.* [9]. Both teams reverse engineered high level views of the WEB application. Our analyses are at a much finer grain, as we need to track individual variable type changes in different execution paths.

In this work we have partially reused the techniques originally developed in [16], complemented by a specific fine-grained analysis and instrumentation oriented to tracking variable dynamic type changes.

C. Type Analysis

Type analysis has traditionally been applied to compiled or interpreted statically typed programming languages such as C++ and Java. See for example [10], [21] for Java programming and the IBM Technical report [11] for C++. Recently researchers have investigated type analyses for weakly typed languages such as JavaScript [20] and dynamically typed languages such as Python [22]. To our knowledge PHP's type system has not been specifically explored so far, however, various kind of taint analyses have been proposed to enforce WEP application security constraints - see for example [33], [17].

Most recently, the Hack programming language has been proposed as an alternative to PHP. Hack is a statically typed version of PHP meant to improve PHP and foster higher quality, safer application development.

Our goal is not to formalize or study the PHP typing system, but rather to evaluate to what extent production PHP developers actually use dynamic typing, and to what extent an application using dynamic typing of variables can be refactored into an equivalent program using static types.

VII. Conclusion

This paper investigates how and to what extent PHP programmers use dynamic typing. Scripting languages such as PHP, Python and JavaScript do not enforce type consistency; variables change type dynamically as they are assigned different values at run time. At a first glance this characteristic seems very appealing, increasing flexibility and easing rapid development. However, it can also make a program much more difficult to understand and thus more bug-prone and difficult to maintain.

For this reason among others, recently Facebook has moved to Hack, a statically typed variant of PHP.

The three research questions investigated in this paper provide evidence that, for four production PHP WEB applications, (1) dynamic typing seems to be used in a relatively limited number of instances (possibly less than 1% of all assignments); (2) most uses of dynamic typing are relatively simple and programmer determination of the type-after can often be determined locally from the source code; and (3) in the majority of cases it is possible to refactor to static typing using local renaming in a relatively easy way.

However, we also found evidence of non-trivial dynamic type changes and at least some cases where simple refactoring can not easily remove the need for dynamic types. These cases often involve library and user defined functions, and specifically functions returning different types on different calls. Removing the dynamic type in these cases would involve refactoring the entire function or library, with possible side effects and a higher cost.

Whether or not Hack and statically typed PHP will be a commercial success remains to be seen. However, it is evident that, at least for the systems we have analyzed (*i.e.*, phpBB2, phpBB3, Drupal and WordPress) dynamically typed variables are relatively rarely used, and in theory these applications could be ported to a statically typed language with relatively modest effort. Indeed, given the low number of assignments involving dynamic typing in these production applications, one may wonder if dynamic typing is needed at all. Nevertheless, we did find cases where it was very difficult to determine the relation between types before and after a PHP assignment statement, and in these difficult to understand cases, the transition to static typing may be more challenging.

Future work will be devoted to refining the dynamic type assignment classification and to providing developers with a tool chain to (1) automatically detect and identify uses of dynamic typing ; and (2) automatically refactor the code to eliminate dynamic typing in the simplest cases. We believe that the more complex cases will be better handled manually by programmers, who can better gauge how and whether the change will be worthwhile.

References

[1] Hack. [Online]. Available: http://hacklang.org/

[2] Hhvm hack. [Online]. Available: http://docs.hhvm.com/manual/en/install.hack.conversion.php

[3] Php manual. [Online]. Available: http://php.net/manual/en/types.comparisons.php

[4] Watir. [Online]. Available: http://watir.com/

[5] (2014) Stack exchange. [Online]. Available: http://arstechnica.com/information-technology/2014/06/why-do-dynamic-languages-make-it-difficult-to-maintain-large-codebases/

[6] (2014) Tiob software. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[7] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Wafa: Fine-grained dynamic analysis of web applications," in *WSE*, 2009, pp. 141–150.

[8] ——, "Automating coverage metrics for dynamic web applications," in *CSMR*, 2010, pp. 51–60.

[9] G. Antoniol, M. Di Penta, and M. Zazzara, "Understanding web applications through dynamic analysis," in *the 12th IEEE International Workshop on Program Comprehension*. Bari, ITALY: IEEE CS Press, June 24-26 2004, pp. 120–129.

[10] B. Bellamy, P. Avgustinov, O. de Moor, and D. Sereni, "Efficient local type inference," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, ser. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 475–492. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449802

[11] P. R. Carini and H. Srinivasan, "Flow-sensitive type analysis for c++," RESEARCH REPORT RC 20267, IBM T. J. WATSON RESEARCH CENTER, Tech. Rep., 1995.

[12] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006.

[13] G. Di Lucca, A. Fasolino, F. Pace, P. Tramontana, and U. De Carlini, "WARE: A tool for the reverse engineering of web applications," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, Budapest, Hungary, Mar 2002, pp. 241–250.

[14] G. Di Lucca, A. Fasolino, P. Tramontana, and U. De Carlini, "Abstracting business level UML diagrams from web applications," Amsterdam, The Netherlands, Oct 2003, pp. 12–19.

[15] ——, "Recovering a business object model from web applications," Dallas, TX, USA, Nov 2003, pp. 348–353.

[16] L. M. Eshkevari, G. Antoniol, J. R. Cordy, and M. D. Penta, "Identifying and locating interference issues in php applications: the case of wordpress," in *ICPC*, 2014, pp. 157–167.

[17] W. Halfond, A. Orso, and P. Manolios, "Wasp: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Transactions on Software Engineering (TSE)*, vol. 34, no. 1, pp. 65–81, 2008.

[18] S. Hanenberg, "An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[19] A. E. Hassan and R. C. Holt, "Architecture recovery of web applications," in *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. ACM, 2002, pp. 349–359.

[20] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Proceedings of the 16th International Symposium on Static Analysis*, ser. SAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 238–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03237-0_17

[21] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX, October 2011.

[22] E. Maia, N. Moreira, and R. Reis, "A static type inference for python," *Proc. of DYLA*, 2012.

[23] E. Meijer and P. Drayton, "Static typing where possible, dynamic typing when needed," in *Workshop on Revival of Dynamic Languages*, 2005.

[24] H. A. Nguyen, H. V. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Output-oriented refactoring in php-based dynamic web applications," in *ICSM*, 2013, pp. 150–159.

[25] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[26] L. D. Paulson, "Developers shift to dynamic programming languages," *IEEE Computer*, vol. 40, no. 2, pp. 12–15, 2007.

[27] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[28] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *Proceedings of the International Conference on Software Engineering*. Toronto, ON, Canada: IEEE CS Press, May 2001, pp. 25–34.

[29] ——, "Understanding and restructuring web sites with ReWeb," *IEEE Multimedia*, vol. 8, no. 2, pp. 40–51, Apr-Jun 2001.

[30] A. Stuchlik and S. Hanenberg, "Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time," *SIGPLAN Notices*, vol. 47, no. 2, pp. 97–106, 2011.

[31] P. Tonella and F. Ricca, "Dynamic model extraction and statistical analysis of web applications," Montréal, QC, Canada, Oct 2002, pp. 43–52.

[32] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, "Variable-precision reaching definitions analysis," *Journal of Software Maintenance*, vol. 11, no. 2, pp. 117–142, 1999.

[33] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *ICSE*, 2011, pp. 251–260.

[34] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.