Identifying and Locating Interference Issues in PHP Applications: The Case of WordPress

Laleh Eshkevari*
Giuliano Antoniol
Department of Génie Informatique et Génie Logiciel
Polytechnique Montréal — Canada
laleh.eshkevari@polymtl.ca
antoniol@ieee.org

James R. Cordy
School of Computing
Queen's University, Canada
cordy@cs.queensu.ca

Massimiliano di Penta
Department of Engineering
University of Sannio, Italy
dipenta@unisannio.it

ABSTRACT

The large success of Content management Systems (CMS) such as WordPress is largely due to the rich ecosystem of themes and plugins developed around the CMS that allows users to easily build and customize complex Web applications featuring photo galleries, contact forms, and blog pages. However, the design of the CMS, the plugin-based architecture, and the implicit characteristics of the programming language used to develop them (often PHP), can cause interference or unwanted side effects between the resources declared and used by different plugins. This paper describes the problem of interference between plugins in CMS, specifically those developed using PHP, and outlines an approach combining static and dynamic analysis to detect and locate such interference. Results of a case study conducted over 10 WordPress plugins shows that the analysis can help to identify and locate plugin interference, and thus be used to enhance CMS quality assurance.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Verification

Keywords

Content Management Systems, PHP, variable interferences, dynamic typing.

*Contact Author

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597153

1. INTRODUCTION

Content Management Systems (CMS) such as WordPress, Joomla, and Drupal are increasingly gaining popularity and changing the way Web sites are developed and evolved. The December 2013 issue of W^{3}Techs[1] reports that about 35% of the 1 million most popular Web sites use a CMS to help authors to manage their contents. The most popular CMS is WordPress (WP)[2] with about 60% of the overall CMS market share and large customers such as Ebay and the New York Times. The massive success of CMSs—and of WordPress in particular—is largely due to their ecosystems of themes and plugins. Themes and plugins allow users to customize the platform's look-and-feel, and to easily develop Web applications providing complex features, ranging from access to social networks to e-commerce features and blogs. As of the end of December 2013, WordPress features 2,177 free themes and 28,546 plugins. Although WP is so powerful, and easy to install and use, often developers have to face unexpected problems with it that are not easily solved.

Motivating scenario. To understand why and where problems arise when using WP, let us go back to mid/end 2012 and consider a scenario in which a user decides to install and run a typical WP-based blog Web site which allows visitors to post and comment on pictures and picture galleries. Let us suppose the user decides to install the WP `NextGEN` plugin, release 1.9.3. `NextGEN` provides an easy-to-configure environment for managing slide shows and picture galleries. The installation task is done with a couple of mouse clicks, and the user may then create her first photo gallery by uploading a few images just to test the presentation and choose a nice page layout. Once done, she may also decide to install the WP `Contact Form 7` feedback plugin and customize a feedback form. To test the form, the user may post a comment on a picture and then approve the comment. All is now ready, but before going live, she runs a scenario combining the two plugins, where she first uploads a set of different pictures into the already existing photo gallery and then mimics the behavior of a visitor commenting on pictures using the form.

[1] `http://w3techs.com/technologies/history_overview/content_management/all`
[2] http://wordpress.org

Suddenly, she bumps into a strange problem: she cannot upload images any more. Unknown to her, she is experiencing a case of *destructive interference* between two (or more) plugins. She remembers that before installing `Contact Form 7` she was able to upload pictures. Indeed, if she removes the `Contact Form 7` plugin, magically she is again able to upload pictures. However, the culprit is not actually `Contact Form 7` – in fact, the problem is due to a wrong file inclusion order in the `NextGEN` plugin! Unfortunately, this problem only appears if the `Contact Form 7` plugin is installed and activated. For this reason, we argue, it was not immediately discovered by `NextGEN` developers, who might easily have attributed the problem to `Contact Form 7` or other plugins with which the problem is manifested. The bug was discovered and documented by the `NextGEN` developers[3], but no solution was offered for users who wish to install both plugins because its origin was unknown.

The general case might be more complex than the simple example explained above. For N installed and activated plugins, one may have to deal with all the possible plugin configurations (installing/activating, and deactivating/removing plugins) to discover the source of an interference.

In order to understand and locate the cause of an interference problem, one must inspect the plugin code and, possibly, examine execution traces. WP and its plugins are written in PHP, mixing the PHP code with HTML, JavaScript and SQL queries, and the user may not have the knowledge to debug such complex multi-tier plugin-based Web applications. Plus, she may not have any way to track down the interferences between the plugins or, even worse, she may not have full access to the server where the Web site is deployed. The only solution available at this stage would be installing/activating or deactivating/removing plugins by trial and error, or browsing mailing lists and forums to look for cases of similar issues to find a suitable solution.

This situation motivates the need for appropriate analysis methods and tools to identify and locate interference problems in CMS such as WP. Manually checking the code and understanding the potential intermixed behavior of thousands of plugins is simply not feasible.

Paper contributions. To deal with some of the aforementioned problems, we first present a taxonomy of WP plugin interferences, and then propose an approach to help WP developers to identify and locate potential plugin interferences.

The approach is able to automatically identify and locate several possible causes of interferences, such as those caused by multiple PHP function definitions or by the WP mechanism used to activate plugin features. We observe that some kinds of WP plugin interferences are caused by the way that WP plugins work, and thus are specific to the WP architecture. However, other sources of interference, such as multiple PHP function definitions, are more general, and thus the problem studied in this paper is not limited to WP and may apply to many other PHP-based Web applications.

To test the feasibility and usefulness of the analysis, we applied our tool chain to two recent WP core framework releases (3.6 and 3.7), augmented with plugins selected to be (1) part of the top 10 must-have plugins, and (2) to have been (possibly) reported to cause interferences. We provide evidence that our approach can effectively help in detecting potential sources of interference and in locating them in the code.

[3]`http://wordpress.org/support/topic/plugin-nextgen-gallery-conflict-with-contact-form-7`

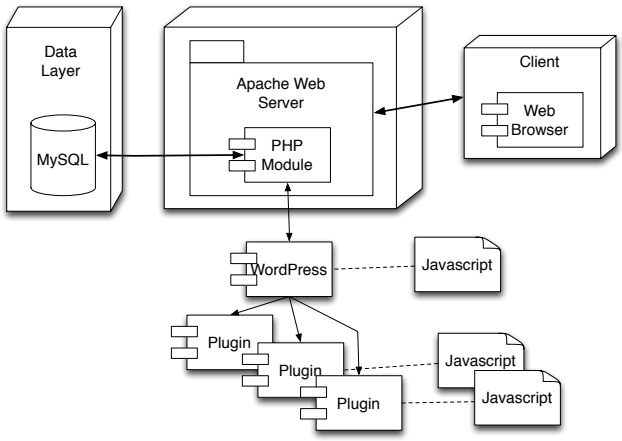

Figure 1: WordPress and LAMP stack.

Paper structure. Section 2 begins by defining the interference problem in WP, and in PHP Web applications in general. Section 3 describes our proposed approach to identify interference problems using a combination of static and dynamic analysis of PHP code. Section 4 reports an empirical study aimed at demonstrating the approach's feasibility and highlighting the relevance of the interference problem in WP. After a discussion of related work in Section 6, Section 7 concludes the paper and outlines directions for future work.

2. INTERFERENCE ISSUES IN WORDPRESS

Figure 1 summarizes the elements of the WP execution environment in the context of the LAMP (Linux/Apache/MySQL/PHP) stack. The PHP Apache module (the interpreter) executes PHP files from inside the Web server. WP uses a relational database in MySQL as a back-end to store transient information, user configuration parameters and various plugin information and parameters. MySQL connections and database operations are handled by the PHP interpreter and thus through Apache. WP is organized into a set of directories; each plugin is stored into its own directory. Several JavaScript components are made available by the core framework, and each plugin can also have its own JavaScript set of libraries.

The WP core framework (release 3.8 – December 2013) is composed of 483 PHP files, 273 JavaScript files and 112 images.

2.1 WP Interferences

In WP additional functionality provided by plugins is added by means of *hooks*, which allow a plugin to 'hook into' the WP framework by having plugin functions automatically called at particular times. There are two kinds of hooks: *filters* and *actions*. Filters, as their name suggests, filter user content transiting from/to the database and the (Web-based) user interface. Actions are functions triggered by specific WP events, such as publishing a page, approving a post, or deleting a user comment.

In addition to filters and actions, a third way for plugins to hook into WP is the *shortcode* API. The shortcode API, available since WP 2.5, is a set of functions for creating macro code for use in WP PHP pages to dynamically call functions, and thus add or modify page contents. For instance, the shortcode `[wp_sitemap_page]` would add the sitemap into a page, while `[slideshow id=1]` generates a slideshow using the first image gallery added to the *NextGen* plugin. In a nutshell, the shortcode API enables plugin developers to create special kinds functions that users can attach to certain pages by adding the corresponding shortcode into the page text.

We classify interferences based on the mechanism causing the interference: shortcode, entity name, variable and database content, and JavaScript. The following subsections provide a succinct description and examples of documented interferences of each kind.

2.1.1 Entity Name Interferences

To enforce information hiding, recent PHP releases implement objects and namespaces. However, namespaces are not used in WP, likely to maintain backward compatibility with earlier Object-Oriented (OO) PHP releases. In a nutshell, PHP has three scope levels: the (global) file-level, the class-level, and the method/function level. This is to say, a file-level function is visible only inside the file and all other files that include the file in which the function is defined. Classes, unless nested inside other classes, are visible at the global (file) scope level.

PHP is a dynamically interpreted language, thus the interpreter has no way to statically verify if at run-time a file will include a function already defined or a class already encountered. Redefinition of functions, classes or variables, defined at the global scope level, may create an interference, *i.e.*, a conflict between an entity already encountered, and a new one of the same name just encountered. WP coding standards[4] suggest that global variables, classes and functions should be named in a way to avoid conflicts, for example by using a unique prefix. Unfortunately, this policy is not enforced, especially in third-party plugins. In other words, a plugin developer or the WP Quality Assurance (QA) team may not be aware of other plugin entity names that create a conflict with a given fragment of code. There is no automatic mechanism to warn of or pinpoint any suspicious or risky naming.

A non-trivial example of potential interference in WP is the presence of two classes implementing the File Transfer Protocol (FTP) and derived from the same base class in the WP core files. One must be very careful to make sure which version is actually instantiated and used. Another example is the class *Services_JSON*, which is defined in both the WP core and the `w3-total-cache` plugin.

2.1.2 Hook Interferences

Hook interference is similar to entity name interference. For filters and actions the interferences will emerge if name of the function to be executed following the event specified by hook_name already exists in WP core or other plugins.

The WP coding standards suggest that shortcode names should be unique, once again such uniqueness is not enforced in any way. Therefore, two different plugins may define the same short-code which may lead to an unexpected behavior.

One well known example of shortcode interference is a conflict between one of the top downloaded WP plugins, *Jetpack*, and the *Contact Form 7* plugin. Among other things, *Jetpack* simplifies form creation, including contact forms. For this reason, early releases of the two plugins used the same shortcode `[contact-form]`. A more recent release of *Contact Form 7* avoids the conflict, by using a different shortcode *i.e.*, `[contact-form-7]`. Nevertheless, the old shortcode `[contact-form]` is still available to ensure backward compatibility, and can still cause a problem.

2.1.3 Variable and Database Interferences

Interference can also occur due to variable definition/redefinition of the same name at the global level, or by defining/redefining constants. This is a subtle and difficult to track kind of interference, because it occurs on the server side, where it is very difficult to debug multi-tier applications. As Figure 1 shows, PHP runs as an Apache module, and the PHP logging levels or Apache error messages are usually not enough to track down the culprit. Of course, it is always possible to instrument and explicitly log variable changes at the server side, but the large quantity of logged information may not be easy to deal with.

WP also stores permanent data into database tables, and the WP coding guidelines advise plugin developers not to create plugin specific tables, if possible, but rather to use a provided API to store key–value pairs in a way similar to a hash table. The API allows creation of an association between a string, the key, and a stored value (or possibly a serialized array of values). In other words, the developer has to define her own key strings to identify the information to be saved into the database and uses the WP API to store/retrieve the information associated with the string. Of course, the string must be unique, otherwise once again a possible conflict with other plugins may occur [5].

2.1.4 Client Side (JavaScript) Interferences

WP and thousands of its plugins rely on JavaScript to implement client-side data validation, Ajax communication, traverse the HTML DOM, or implement various other kinds of client-side functionality. Unfortunately, it may happen that two different plugins, activated in the same client page, load incompatible JavaScript code or different, incompatible, releases of the same JavaScript library.

An example of this kind of interference can by caused by different versions of the JavaScript library *JQuery* [6]. *JQuery* simplifies HTML document traversal, manipulation, event handling, and animation. The WP core framework already includes a *JQuery* release; however, several plugins include their own, different, *JQuery* release.

Let's consider the following scenario. To develop a Web site, the user uses the *JQuery Superfish* menu plugin. Also, let us assume that the WP installation also includes the *Contact Form 7* plugin. Unfortunately, the version of *JQuery* used by *Contact Form 7* may interfere with other versions of *JQuery* used by WP or by other plugins. Such interference often occurs while WP is loading the page header. Depending on

[4] `hhttp://make.wordpress.org/core/handbook/coding-standards/`

[5] `http://wordpress.org/support/topic/fields-not-saved-in-database`

[6] `https://jquery.org/`

the release of the *Contact Form 7* plugin, the loaded *JQuery* version may or may not cause conflicts [7].

The easiest solution is to make sure the required JavaScript release is loaded, and possibly to de-register *Contact Form 7* JavaScript. However, this does not guarantee the *Contact Form 7* plugin will work properly.

2.2 Errors, Faults, and Failures in the Context of Interferences

Plugin destructive interferences occur if and only if interfering plugins are installed and activated, and a certain sequence of events occurs. Strictly speaking, one should make a distinction between the developer **error** (*e.g.*, a poor shortcode naming choice), the actual interference **failure** (*e.g.*, a Web page not created), and the **fault** (*e.g.*, the chosen conflicting shortcode). If the fault is not executed the failure is not observed. Our goal is to detect and pin down potential interferences, and thus to detect potential faults. We believe it is safer to warn the user of a potential conflict and let the user decide if the code should be modified or not.

Let us return to the example interference described in the introduction of this paper. The order in which *NextGEN* loads configuration parameters – *i.e.*, the order of its include files – first causes the loading of a regular user profile, then it unsets this user profile and loads an administrator profile. Here the developer error resulted in a fault which is an incorrect order of file inclusion. This results in a mixture of normal user and administrator privileges. For example, it defines the constant `WP_ADMIN` as true (line 14 of the erroneously-loaded `wp-admin/admin.php` file). The failure is experienced if *Contact Form 7* is installed and activated. To detect the fault we need do nothing specific about the particular *Contact Form 7* plugin – it suffices to simply track the file order of inclusions. To fix this problem, the order of includes must be changed [8].

One may argue that the actual interference is due to a variable being inconsistently reassigned (*e.g.*, `WP_ADMIN`) which causes the actual failure. However, in the specific case, the real fault is incorrect include order which, incidentally, is easier to track and detect.

3. PROPOSED APPROACH AND TOOL SUPPORT

The approach for identifying interferences in CMS developed with PHP is based primarily on a static analysis, complemented by dynamic analysis to track the run-time resolution of includes. It is important to note that the proposed approach can also be applied using the static analysis only. This would be especially advantageous to perform a cheap and fast analysis without the need for deploying and executing the CMS being analyzed.

To analyze PHP source code, we rely on two widely-used infrastructures, the PHP parser from the Eclipse PHP Development Environment (PDT)[9], and TXL[4]. Specifically, we use the Eclipse PDT parser to assist in the static analysis, and the TXL source transformation engine to add the code instrumentation needed to collect information at run-time.

[7] `http://stackoverflow.com/questions/4964068/javascript-conflict-contact-form-7-superfish-js`
[8] `http://wordpress.org/support/topic/plugin-nextgen-gallery-conflict-with-contact-form-7`
[9] http://projects.eclipse.org/projects/tools.pdt

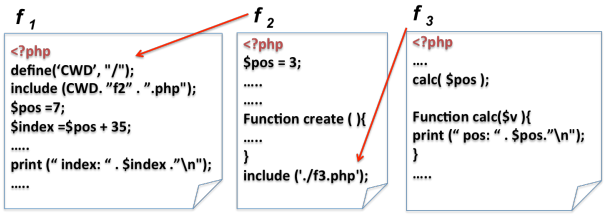

Figure 2: Example of include in PHP.

3.1 Analyzing PHP data flow and includes

The goal of our analysis is to extract information needed to perform the interference analysis detailed in Section 3.2.

First, we use the PDT parser to create an Abstract Syntax Tree (AST) for each PHP file in the WP directory tree. We build a symbol table to collect information about PHP entities namely files, classes/interfaces, methods/functions, method/function parameters, fields, constants, and local variables.

3.1.1 The PHP Include Mechanism and Variable Declaration

Before performing deeper analyses such as data flow analysis, we need to locate and resolve includes. Differently from C/C++, where the name of the file to include must be contained in a string literal (possibly built at compile-time via pre-processor)—*i.e.*, #include "filename"—in PHP the inclusion is done with the `include` statement that accepts as a parameter an expression. There are no constraints on the include expression which can contain variables and calls to functions as well as string operators. In other words, often, the file name path is dynamically computed and built at run-time. To avoid circular inclusions, PHP provides different include statements, namely `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 included already, *i.e.*, avoiding a multiple inclusion.

Consider the example in Figure 2. The PHP `define` statement takes two arguments: a string identifying the constant to be defined, and an expression to be evaluated and assigned to the constant (the right hand side). In Figure 2 the constant `CWD` is defined in f_1 by the statement `define( 'CWD', '/' )` and `'/'` is its value. In the same example we see file f_1 include file f_2 through the include statement `include (CWD."f2".".php")` where the path to the file to be included is the result of the concatenation of three strings *CWD*, *"f2"*, and *".php"*, where the constant `CWD` is already defined in previous line. Moreover, If included files are not resolved, one would think that variable *pos* is first encountered and thus defined in file f_1, while actually it is declared/defined in file f_2.

3.1.2 A Fixed-Point PHP Include and Data Flow Analysis Approach

Algorithm 1 presents a high-level view of the computation performed to resolve includes and constants. First, all files in the system are traversed and entities of interest are

added to the symbol table. We identify the WP API calls used to instantiate hooks and short-codes by collecting the `add_action`, `add_filter`, and `add_shortcode` function calls and save the needed information in a separate data structure which will be used for the analysis of hook interferences. For each entity we extract its line number, name, signature (if exists), type (if exists), and scope. For constant entities we collect their values (*i.e.*, right end side) if not dynamically built at run-time. If the value is not a scalar but it is a concatenation of strings and/or return value of functions, we collect the whole statements and mark the entity as non resolved. In the same way we process the include statements (`include`, `include_once`, `require`, `require_once`) as well. We store in the symbol table structure also the statement parse tree, later used by the symbolic execution. To statically resolve constant and include statements, we implemented a simplified symbolic execution. The simplified symbolic execution implements the behaviour of two PHP magic variables and two functions as well as the string concatenation operator (See Table 1). We rely on API function provided through `FileNetworkUtility` in PDT to handle the access to the parent (current) directory (*e.g.*, "../" and "./")

Once files are traversed and the symbol table initially populated, the fixed-point algorithm attempts to resolve as many include file paths as possible. Before this step, we apply a simple heuristic using the function *ScanAndAssignNameIfUnique*. It may happen that the include path ends with a file name (*e.g.*, `'ABSPATH' . ngg-config.php`) and it may happen that this file name is actually unique and not the suffix of any other file. In such cases, there is no need to perform any complicated calculations and the include relation is resolved based on the file name identity.

Next, as shown in Algorithm 1 if the current entity is an include or a constant, and (so far) it has not been resolved *i.e.*, the value is not known, the function *SymbExecAndUpdate* attempts to compute the current define right end side or include parameter value. If *SymbExecAndUpdate* succeeds, it returns true so that the collected new information can percolate and improve collected information, as well as updating the values of other entities depending to the current entity. If *SymbExecAndUpdate* fails, the entity will be marked as delayed, and its value may be resolved in following passes. In fact, if a new include file path (or constant) is resolved this may impact on variable definition as well as on other constants or includes.

Algorithm 1 terminates when it is not able to further update any information (*i.e.*, when it has reached a fixed-point). In general, it may happen that, at this stage, some constants or includes are still not resolved, because the path of some files is still unknown. This typically happens for two reasons: (i) the right-hand side of the string constant definition or the include parameter contains one or more PHP variables; or (ii) the string value is obtained from a user defined function.

Table 1: Implemented functions, operators and Magic Variables

Function/ Operator/ Variable	Description
dirname	Extract directory name
basename	Name with any leading directory components removed
.	Concatenate two strings
__FILE__	Contains the current file absolute path
__DIR__	Equivalent to dirname(__FILE__)

Algorithm 1 Include resolution and symbol table construction.

```
update=false
for file in System do
    entities = extractEntities()
        addToSymbolTable(entities)
end for
for ent in SymbolTable do
    if ent == Const|ent == Include&¬ent.resolved then
        ScanAndAssignNameIfUnique(ent)
    end if
end for
repeat
    for ent in SymbolTable do
        if ent == Const|ent == Include&¬ent.resolved then
          update| = SymbExecAndUpdate(ent.stmnt, SymbolTable)
        end if
    end for
until update
```

The remaining unknown includes make the analysis imprecise. In general, the way the analysis is performed avoids the presence of false positives, but cannot exclude false negatives. Typically—as it also happened in our study (see Section 4)—most includes are statically resolved even before the fixed-point algorithm, and thus in principle, at least for WP, a good quality analysis can be done without the need for dynamic analysis. However, if a complete analysis is needed, the only possible solution is to resort on dynamic analysis to track the remaining unresolved relations. Clearly, dynamic analysis requires deploying and executing the application, and the design of scenarios that exercise the unresolved includes.

Once the includes have been resolved, we build a directed graph for the whole WP application, including plugins. Nodes (f_i) of the graph correspond to files, and an edge from f_i to f_j indicates that f_i includes f_j. if file f_1 includes files f_2. Edges are labeled with the specific type of include (*i.e.*, `include`, `include_once`, `require`, `require_once`), with the line number where the include statement occurs, and the scope (file, class, function/method) to help distinguishing cases where the include is inside a function/method declaration, class declaration, or file. This include graph is used by the steps defined below and to check certain kind of includes faults and include oddities (*e.g.*, the same file included in multiple locations in the same file).

3.1.3 Extracting variable declarations

The include graph is used to perform another pass over the AST of all files to extract the first definition of a variable in a file as its declaration. The include graph built in the first pass makes it possible to verify if variable under analysis belongs to the current file or another files being included. At this point, we just go one level up, *i.e.*, for variable `$pos` that is defined in f_1 we check if the file that is included immediately before the definition of the variable f_2 has a definition for variable `$pos`, or files that include f_1 have a definition for `$pos`. If no other file has a definition for `$pos`, then we add `$pos` to the symbol table related to the scope of file f_1. In the example of Figure 2, the variable `$pos` is not added to the symbol table under the scope of filef_1 as it belongs to file f_2.

It is important to point out that the extraction of variable declarations is not 100% precise, as our analysis is both context and flow insensitive, *i.e.*, the order of statements and the call sites of methods are not taken into consideration. In other words, our analysis could interpret that both files f_1 and f_2 declare variable `$v`, while in reality the variable belongs only to one of those files.

3.1.4 Extracting def-use pairs

In the final pass over the AST, we collect the definition and use statements. For each entity in such statements, we first look up the entity in the symbol table of the current file. If such an entry (*i.e.*, a declaration) is found, we keep track of the use/definition statement in a proper data structure. Otherwise, using the include graph built in the previous step we look into the symbol table for files that have include relation with the current file and find an entity declaration that corresponds to the entity being defined or use in the statement under analysis, and add the def-use statements to the data structure. Again, because we are both context- and flow insensitive, definitions and uses are a conservative super-set approximation of the actual information.

3.2 Detecting Interferences

Based on the information collected using the approach described in Section 3.1, we detect a subset of the different kinds of interferences introduced in Section 2. We are able to detect conflicting entity names, hooks, and database code, as well as certain include patterns documented to create potential interference. Some specific kinds of interferences—namely those related to Javascript—are left for future work, because they require a much heavier combination of static and dynamic analysis (*e.g.*, the Javascript code may be automatically generated from the server side). In the following, we explain how each kind of interference is detected.

3.2.1 Detecting Conflicting Entity Names

To identify conflicting entity names, we rely on the symbol table. We traverse the symbol table, and collect class and function names, and compare for identical names.

3.2.2 Detecting Conflicting Hooks

Detecting conflicting hooks and short-codes is less straightforward than detecting conflicting entity names. We first search for the WP API calls used to instantiate hooks, namely functions `add_shortcode`, `add_action`, and `add_filter`. For all three functions we evaluate the first parameter (the name of the hook or short-code). If the first parameter is a constant string, we stop, if it is a variable we check where the variable is defined and if it is defined by a constant string. We also search for defined constants used in the hook registration API. These are collected as possible hook and short-code names and used to compare with other plugin hooks and short-codes.

3.2.3 Detecting Conflicting Database Code and Conflicting Variables

Similarly to detecting conflicting hooks and short-codes, we attempt to track strings used in the WP database API for adding plugin data to the database. Two functions are specifically of high interest:

- `add_option` is used to add a named option/value pair to the options database table. The function accepts four parameters, where the first parameter refers to the name of the option be and the second parameter is the corresponding value. An attempt to add an already existing option to the data base will be failed. The name of the option should be all lowercase string and words should be separated by underscore.
- `add_post_meta` is used to add custom fields to a post. The function accepts four parameters: the postid, the key of the custom field, its value and a boolean value determining whether or not the key should be unique.

As for hook interferences, here we check the first parameter of `add_option` and the second parameter of `add_post_meta`. Interference is detected when two or more different plugins call these functions with same name or custom field key.

3.2.4 Detecting Risky Include Configurations

WP plugins should respect coding standards as well as certain include patterns. Given the include graph, it is possible to check for include oddities and for specific risky include patterns. Include oddities are cases in which a file including another file for multiple time. Risky includes are cases in which files from plugins including files from the WP core as this may generate unwanted side effects and interferences.

For example, in WP 3.7, `admin.php` includes the files `admin-footer.php` and `admin-header.php` in all but one branch of an if statement. This can be easily refactored in a more easy to read and understand way by pulling out of the if the two includes. Another example is the file `sitemap-core.php` in the latest release of plugin `google-sitemapgenerator`, where class `classsnoopy.php` of the core WP is included. Future changes to functions of the class `classsnoopy.php` will directly affect the `sitemap-core.php`, and may not be noticed since the PHP interpreter does not statically verify whether a function invocation has a valid definition.

4. EMPIRICAL STUDY

The *goal* of this study is to use the approach described in Section 3, with the *purpose* of detecting potential interferences in WP with a set of installed plugins, with the *purpose* of understanding the relevance of interference problem and assessing the capabilities of the proposed approach to deal with it.

The *context* of the study consists of WP itself—in two releases, namely 3.6 and 3.7—along with a set of 10 installed plugins. We did not consider the most recent versions of WP (*e.g.*, 3.8) because many plugins were not tested with that release. For what concerns the selection of plugins, we focused on 10 popular ones. There is no general consensus on the top ten must-have plugins; depending from the Web site's application domain, user communication goals and project constraints, different plugins may better serve the Web site's objectives. Indeed, there are several different lists of the top ten must-have WP plugins on the Web. Based on the WP most popular plugins[10]and other three most popular lists, namely WeDesignPixel [11], TreeHouse Blog [12]

[10]http://wordpress.org/plugins/browse/popular
[11]http://wedesignpixel.com/top-must-have-wordpress-plugins
[12]http://blog.teamtreehouse.com/best-free-wordpress-plugins-for-common-website-functionality

and Selz Blog [13], we have chosen the set of plugins reported in Table 2. As the table shows, we performed our study on two configurations of WP: (i) an old one, consisting of WP 3.6 plus some old versions of the 10 plugins, and (ii) a more recent one, consisting of WP 3.7 plus some newer versions of the plugins.

4.1 Research Questions

The goal of the study is to address the following research questions.

- **RQ$_1$**: *How many potential interferences can we detect using static analysis?* The aim of this research question is to assess the relevance of the investigated problem, *i.e.*, to investigate to what extent the interferences illustrated in Section 2 are common in WP and in its plugins. Other than investigating how many interferences can be detected, we also perform a qualitative analysis on some of the interferences, highlighting cases where the interferences have been documented, *e.g.*, in WP forums.

- **RQ$_2$**: *How effective is the static analysis in detecting interferences?* As explained in Section 3, our approach is mostly based on static analysis, however it can be complemented by dynamic analysis to resolve some includes that cannot be resolved statically. The goal of this research question is to investigate on the added value of dynamic analysis. Our conjecture is that, should this added value be fairly limited, then developers could just rely on static analysis. This is because the latter is cheaper, and does not require deploying and executing the system under analysis.

It is important to point out that our empirical evaluation does not focus on the approach's accuracy/precision. Our approach, by construction, does not produce any false positives. The approach goal is to detect the fault *e.g.*, two different functions or classes with the same name, or two hooks of two different plugins installing different handlers with the same short-code string. As in testing, if the fault is not executed the failure is not observed. Thus the fact that, for example, two functions have the same name not necessarily manifest itself into an interference. In a nutshell, every time an interference is detected, this, is, indeed, a "potential" problem: that is, static analysis is able to identify potential problems in the code. However, as is the case for any static analyzer—including for example vulnerability detectors—it may or may not happen that there exist execution scenarios in which the fault is produced and the failure will be manifested.

Instead, the approach may suffer from false negatives, *i.e.*, there may be interferences that the proposed method is not able to detect. Section 4.3 will discuss such limitations in detail, by also providing some examples of interferences we failed to detect.

4.2 Study Results

WP and plugin source code was analyzed, information extracted and potential problems identified. This section reports quantitative results on the discovered interferences aiming at answering the two research questions formulated above. All potential interferences are available for download on-line[14].

Specifically, in the following we quantify potential interferences found for the categories defined in Section 2 and provide discovered examples of such interferences. After that, we summarize answers to our research questions.

Include-based Interferences:

Include relations are computed as described in Section 3 and modeled as a graph. Graph node represent PHP files, and edges represent include relations. Edges are labeled with the line of code at which the include is encountered and the type of inclusion (*i.e.*, `include`, `include_once`, `require`, and `require_once`). Edges are also labeled by the entity performing the include *i.e.*, a class or method, or a function. Given the large number of inclusions, manual inspection of such graphs is not practical even for a single plugin. Therefore, although we will show some excerpt of graphs as examples, such graphs are automatically analyzed to detect possible interferences. Table 2 reports a summary of the include relations in both releases of WP and their corresponding plugins.

To simplify the computations we assume that we know the values of the constants: `ABSPATH` (the path where WP is installed), `WPINC` (location of include directory), and `WP_PLUGIN_DIR` (location of plugin directory). As Table 2 shows, most of WP includes are resolved before the fixed point step; this points to a disciplined and not overly complex include file regimen in the core framework. However, for plugins the situation changes. Consider the plugin `W3 total cache`. This plugin has almost the same number of include relations as WP itself, but almost half of the included file relations are resolved only by the fixed point step. On summary, only 8.2% (number of unknown after fix point over the total number of include relations) cannot be identified statically.

When looking at release 3.7 of WP and its related plugins, the situation does not change dramatically, and overall only about 9% of the include relations are not statically computed.

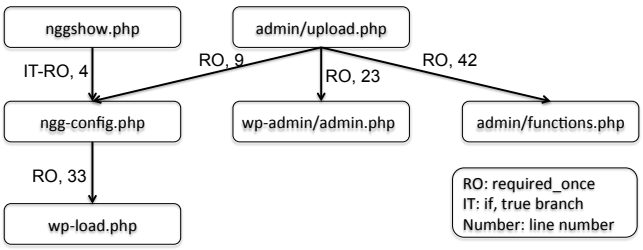

Figure 3: NextGen Dynamic Include Graph Fragment.

When a plugin is started, WP is already loaded and running thus a plugin should not include the top level WP startup file. To be on the safe side, we encoded a simple rule that says: "a plugin should never load WP load file *wp-load.php*".

In addition, one may wish that the include structure is simple with easy to follow include relations. We decided to flag as odd include statements, statements including the same file in different branches of an if, or include potentially

[13]https://selz.com/blog/10-must-wordpress-free-plugins-starting-online-business-selling-digital-downloads/

[14]http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/wp-data.tgz

Table 2: Analyzed releases of WP and its plugins, with details about include relations.

	Old Release						New Release					
	Rel.	Includes	Unknown Before Fix Point	Unknown After Fix Point	Edges	Nodes	Rel.	Includes	Unknown Before Fix Point	Unknown After Fix Point	Edges	Nodes
WordPress	3.6	629	37 (6%)	35 (5%)	627	366	3.7	647	37 (6%)	35 (5%)	645	368
NextGen Gallery	1.9.3	114	26 (23%)	12 (10%)	114	71	2.0.40	144	37 (26%)	22 (15%)	144	149
Google XML Sitemap	3.2.7	5	2 (40%)	2 (40%)	5	7	3.3.1	5	2 (40%)	2 (40%)	5	7
Contact Form 7	3.2	16	15 (94%)	4 (25%)	16	19	3.6	19	18 (95%)	5 (26%)	19	20
Akismet	2.5.6	3	0 (0%)	0 (0%)	3	4	2.5.9	3	0 (0%)	0 (0%)	3	4
SEO by YOAST	1.1.7	22	18 (82%)	2 (9%)	22	22	1.4.22	42	35(83%)	2 (5%)	42	40
WP Sitemap Page	1.0.12	1	1 (100%)	1(100%)	1	2	1.0.12	1	1 (100%)	1 (100%)	1	2
Google XML Sitemaps for qTranslate	3.2.7.1	6	2 (33%)	2 (33%)	6	8	3.3.1	6	2 (33%)	2 (33%)	6	8
YARPP	3.5	17	16 (94%)	5 (29%)	17	17	4.1.1	26	23 (88%)	4 (15%)	26	25
Jetpack	2.7	95	37 (39%)	15 (16%)	95	108	2.3.5	126	63 (50%)	20 (16%)	126	143
W3 Total Cache	0.9.2.4	592	335 (56%)	45 (8%)	593	332	0.9.3	436	168 (38%)	33 (7%)	436	299
Total		1,500	489 (33%)	123 (8%)	1,499	956		1,455	386 (26%)	126 (9%)	1,453	1,065

leading to future unwanted side effect if the include file will be modified.

We found 35 and 50 cases of possible risks in WP 3.6 and WP 3.7 respectively, and 290 and 247 cases of oddities in WP 3.6 and 3.7 respectively.

In following we describe one such instance of risky file inclusion in plugin `nextgen-gallery`. We were aware of this conflict and thus it can be considered a sanity check.

When the `nextgen-gallery` plugin is started it requires the file *ngg-config.php*, which wrongly includes the WP file *wp-load.php*.

As shown in Figure 3, *wp-load.php* causes the loading of the admin bootstrap *wp-admin/admin.php* in administrator mode. However, the latter was previously loaded in non-admin mode. To fix the problem one has simply to comment out the include of *wp-load.php* and add the include of *wp-admin/admin.php*, thus clearly separating administrator mode from non administrator mode.

The simple rule "a plugin should never load `wp-load.php`" correctly allowed us to identify the `nextgen-gallery` documented problem. This rule correctly identifies the `nextgen-gallery` issue. A similar issue occurred in `W3 total cache` (again WP 3.6). The file `min/index.php` at line 13, requires the loading of `wp-load.php` in a way similar to `nextgen-gallery`. That is to say, there is currently no interference in `W3 total cache`, but rather just a risky situation, in which a non-expert might easily extend the plugin to create a problem similar to the one occurred in`NextGEN`.

Table 3: Name interferences in WP 3.6 and top 10 plugins.

Entity Type	same plugins	different plugins	plugins and cores	core
Class/interface	30	28	6	4
Functions	8	0	36	64
Constant	7	18	18	47

Table 4: Name interferences in WP 3.7 and top 10 plugins.

Entity Type	same plugins	different plugins	plugins and cores	core
Class/interface	28	28	0	4
Functions	14	0	34	64
Constant	16	18	4	49

Table 5: Hook interferences in WP 3.6 and 3.7.

Program	actions	filters	short-code
WP 3.6	33	44	5
WP 3.7	38	42	2

Entity Name Interferences: Table 3 summarizes the name interferences occurring in WP 3.6 and in the top 10 plugins with releases shown in the second column of Table2. Table 4 shows the name interference in WP 3.7 and the top ten plugins with releases shown in the second column of Table2. For example, we found two class definitions `soap_fault` in the old and new releases of plugin `w3totalcache`. Both classes are kept because of backward compatibility and they both do not have class body. These classes belongs to an inheritance hierarchy that is also duplicated. Strangely enough the implementation of the root classes (`nusoap_base`) of these two hierarchies is essentially identical.

An interesting example is the duplicate definition of method `wp_cache_add_global_groups` with same body implementation of WP core and plugin `w3totalcache` in the old and new releases. Another example is a name interference between a method `safecss_filter_attr` in WP core and `jetpack` plugin, where the implementation of methods is not the same.

Hook Interference: Table 5 reports all the interferences occurring in WP 3.6 and WP 3.7. For example, method `module_toggle` of class `Jetpack_Post_By_Email` will be called in plugin `jetpack` when both actions `jetpack_activate_module_postbyemail` and `jetpack_deactivate_module_postbyemail` are activated. Although the action names suggest opposite semantics, the same method will be called.

Another example is the filter `the_title`, where three functions `esc_html`, `post_title` and `esc_html` are being called when the filter is activated. The first and the third filters call the same function `esc_html` defined in file `formatting.php` in WP core include directory while the second filter calls function `post_title` defined in class `customcss.php` of plugin `jetpack`.

Database Interferences: In total, we found 35 and 38 calls to the WP 3.6 and 3.7 API functions for adding plugin data to the database. We only found two cases of interferences between plugin `google-site-map` and `google-xml-sitemaps-v3-for-qtranslate` in both new and old release. Both plugin use the WP API function `add_option` to save the page through `sm_cpages` key to the option table in database.

Table 6: Unknown includes resolved in WP 3.6 and its plugins by means of some dynamic analysis.

Scenario	Discovered Unknown	%
1	17	(13%)
2	9	(7.3)%
3	31	(25%)
4	13	(10%)
5	12	(9.7%)
Overall	33	(26%)

As these two plugins are almost the same, it is not surprising to find such interference.

Summary:

In summary, we can answer to our research questions as follows:

- **RQ1:** Using static analysis we can find potential cases of interferences in WP. However, only the developers can confirm whether or not such interferences are indeed real threats.
- **RQ2:** Static analysis only fails to detect 9% of the include relations between the files. However, this in turn could affect the completeness of hook and database interferences.

4.3 Discussion: Approach Limitations and Documented Interferences

The resolution of file inclusion is one key component of our analysis. If includes are not correctly resolved, then hook and database analysis will be directly affected.

Although, as explained, we used static analysis to resolve includes, we have validated it by means of dynamic analysis. Clearly, a thorough dynamic analysis would have required to exercise all possible paths of the Web application that alter the values of include file names. In our analysis, we focus on five execution scenario, to see how we could have discovered some includes not resolved statically.

By comparing the log of dynamic analysis and the include relations that are extracted statically we identify s subset of unknown relations. Table 6 indicates the number of unknown includes that were discovered by these five scenarios in WP 3.6 and its corresponding plugins.The second column of the table shows the number of unknown files that are resolved dynamically. Overall, by executing these scenarios 26% of all unknown include relation were identified.

Table 7 summarizes all cases in which the static analysis analysis fails. In some cases the include could not resolved because the string passed as a parameter to the `include` function was produced as output of a function call, and the simplified symbolic execution could not resolve that. In other cases, the file name could not be produced because this would have required a symbolic execution able to support a context- and flow-sensitive data flow analysis. In summary, very likely a more sophisticated symbolic executor could have further improved the completeness of the include resolution. However, since the unknown includes represent only about 9% of the total include relations, a lightweight analysis like the one we proposed is appropriate and able to scale up to the size and complexity of WP with its plugins.

We found four cases of such interferences discussed online in WP forums:

Table 7: Where static analysis fails.

Program	Path contains variables	function calls
WP 3.6	73	12
WP 3.7	89	8

- short-code `[contact-form]`: the interference between the plugins `jetpack` and `contact-form-7`[15].
- short-code `[slideshow]`: the interference between the plugins `jetpack` and `nextgen-gallery`[16].
- short-code `[audio]`: the interference between the WP core and plugin `jetpack`[17].
- risky include: the interference between plugin nextgen-gallery and `contact form 7`[18].

Moreover, we found a discussion on WP from regarding the interference between WP core 2.9 and plugin `Twitter Tools` regarding the duplicate definition of class `Services_JSON`[19]. Plugin`Twitter Tools` is not among the 10 top plugins we analyzed, but we found the same duplicate of class definition between WP core 3.6 and old release of plugin `w3-total-cache`. It is interesting to note that such duplicate of class `Services_JSON` does not exist in WP 3.7 and the new release of plugin `w3-total-cache`. WP documentation of short-code discourages developers in using hyphens (dashes) in the names of short-code, but never enforce such policy. In total, we found nine cases of short-code names with hyphens in both WP and plugins, one such short-code is the problematic `[contact-form]` discussed earlier.

5. THREATS TO VALIDITY

This section discusses the threats to validity that can affect the results of our study. 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 4.3, the only imprecision that could have occurred in our analyses is related to incorrect identification of include relations. However, to check the validity of the includes obtained by means of static analysis, we have performed a dynamic analysis, where include file names were generated while executing certain scenarios. Such analysis confirmed the correctness of the identified includes.However, since not all includes were resolved by static analysis, this could have limited the completeness of the identification of other kinds of interferences. For false negatives, as said a thorough analysis is not possible because it requires a deep knowledge of WP and of their

[15]http://contactform7.com/jetpack-overrides-contact-forms/

[16]http://wordpress.org/support/topic/plugin-nextgen-gallery-shortcode-slideshow-conflicts-with-jetpack-by-wordpresscom-plugin

[17]http://wordpress.org/support/topic/plugin-jetpack-by-wordpresscom-audio-shortcode-is-not-falling-back-to-flash-in-ie9-ff

[18]http://wordpress.org/support/topic/plugin-nextgen-gallery-conflict-with-contact-form-7

[19]http://wordpress.org/support/topic/plugin-twitter-blackbird-pie-cannot-redeclare-class-services_json-on-line-116

plugins. However, at least we found confirmation four cases of potential interferences being documented on WP forums. In future we plan to conduct a survey involving WP and plugin developers to verify the severity of the interferences we found in these programs.

External validity threats concern the generalizability of our results. The study is limited to two releases of WP, and to a subset of its plugins. Although we expect that similar problems can occur with other plugins and, possibly, with other PHP frameworks besides WP, further, larger studies need to be conducted to verify such a conjecture.

6. RELATED WORK

This section describes related work concerning the analysis of Web applications and, specifically, of PHP applications.

Reverse engineering of WAs requires static and dynamic analysis and several works have contributed to the advancement of the fiend since early 2000. The first significant contribution was given by Ricca and Tonella, who developed the *ReWeb* tool to perform analyses on web sites [12, 13]. In particular, Ricca and Tonella introduced a graphical representation of the web site to extend to WAs traditional static flow analyses such as reachability, dominance, and data flow analysis. *ReWeb* performs the mentioned analyses plus it detects navigational patterns. Ricca and Tonella also proposed to enhance static analysis by using dynamic information [14].

Di Lucca *et al.* [5, 6, 7] 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.

Hassan and Holt [8] proposed an approach to automatically recover the architecture of Web applications. Their approach is able to recover the interaction between various components of a Web application—including its presentation, business logic, and database—by using specialized connector extractors.

Antoniol *et al.* proposed WANDA [3] an approach and a tool that instruments Web applications and combines static and dynamic information to recover the as-is architecture and, in general, the UML documentation of the application itself, including static (class, object) and dynamic (sequence) diagrams.

We share with the aforementioned contributions the idea of using static analysis—and when needed also dynamic analysis and dynamic analysis—to analyze Web applications. However, rather than being interested to recover the architecture of a Web application, we have a specific goal, *i.e.*, to detect certain types of interferences.

In recent years, work has been done to develop analyzers specialized for PHP applications. Nguyen *et al.* [9] 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. WebDyn accurately performed output-oriented refactoring in all four real word Web application they studied.

Nguyen *et al.* [11] proposed an automatic approach, DRC, to identify dangling reference errors in PHP programs using static analysis of source code. DRC applies symbolic execution of PHP programs to identify variable declarations and references. For each detected declaration or reference, DRC associates it with the current path constraint of the symbolic execution. To identify the declaration and references of entities embedded in HTML or SQL script within the PHP code, DRC uses the tree-based representation, called D-model [10]. Next, for all variable references, DRC identifies a declaration that matches the reference. While in principle a thorough symbolic execution as the one proposed by Nguyen *et al.* could also be applied in our case, we have opted for a lightweight approach where a (simplified) symbolic execution is only used for the purpose of resolving include. Albeit lightweight, our analysis was able to cover over 90% of the include relations.

Alalfi *et al.* [1] also proposed an approach based on dynamic information to analyze Web applications, and they suggested the use of coverage metrics [2] to collect accurate information. They applied this approach for the purpose of security analysis. Rather than using dynamic analysis, our approach applies a lightweight approximated static analysis, complemented, if needed, by dynamic analysis having the sole purpose of analyzing include relations not identified by the static analysis.

7. CONCLUSION

Content Management Systems (CMS) are increasingly used to ease the development of Web sites. WordPress (WP) is the most popular CMS, with about 60% of the CMS market share and millions of Web sites relying on its functionality and its rich plugin ecosystem. This paper presented a taxonomy of potential plugin interferences in WP, and an approach to support developers in predicting, identifying and locating potential interferences before they become a real problem. To demonstrate the feasibility of our approach, we applied it two WP releases extended with 10 among the most commonly installed WP plugins.

The proposed approach allowed us to detect both documented interferences and other possible sources of potential interference. Noticeably, in most cases static analysis was just sufficient. This means that in most cases the proposed approach can be applied without the need for deploying and executing the Web application.

Future work will be devoted to improving the static analysis as well as other kinds of interferences, such as JavaScript interferences. Finally, we would like to extend the analysis to a large number of plugins and to other CMS and PHP Web applications besides WP, and discuss our findings with WP developers.

8. REFERENCES

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

[2] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Automating coverage metrics for dynamic web applications. In *CSMR*, pages 51–60, 2010.

[3] G. Antoniol, M. Di Penta, and M. Zazzara. Understanding web applications through dynamic

analysis. In *the 12th IEEE International Workshop on Program Comprehension*, pages 120–129, Bari, ITALY, June 24-26 2004. IEEE CS Press.

[4] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[5] 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*, pages 241–250, Budapest, Hungary, Mar 2002.

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

[7] G. Di Lucca, A. Fasolino, P. Tramontana, and U. De Carlini. Recovering a business object model from web applications. pages 348–353, Dallas, TX, USA, Nov 2003.

[8] 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*, pages 349–359. ACM, 2002.

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

[10] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-locating and fix-propagating for html validation errors to php server-side code. In *ASE*, pages 13–22, 2011.

[11] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Drc: a detection tool for dangling references in php-based web applications. In *ICSE*, pages 1299–1302, 2013.

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

[13] F. Ricca and P. Tonella. Understanding and restructuring web sites with ReWeb. *IEEE Multimedia*, 8(2):40–51, Apr-Jun 2001.

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