UNIVERSITY OF AMSTERDAM

Reducing Dynamic Feature Usage in PHP Code

by

Chris Mulder

A thesis submitted in partial fulfillment for the
degree of Master of Science Software Engineering

in the
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Centrum voor Wiskunde en Informatica

Saturday 31st August, 2013

UNIVERSITY OF AMSTERDAM

Abstract

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Centrum voor Wiskunde en Informatica

Master of Science

by Chris Mulder

Most PHP applications make use of the language's dynamic features. Even though these features give the developers great flexibility, they can make static analysis impossible and can have a negative impact on performance. Using dynamic analysis, we were able to transform occurrences of dynamic feature usage to static code with the same semantics. This transformed code enables more accurate static analysis, however we did not measure any performance improvements.

Acknowledgements

First of all I would like to thank my two tutors: Mark Hills and Jurgen Vinju. Without their feedback and ideas I would not be able to write this thesis. Next them I want to thank Frank Tip for giving me some pointers concerning dynamic tracing.

I finally would like to thank my girlfriend, family and friends for there mental support during this intensive period.

Contents

Chapter 1

Introduction

1.1 PHP

PHP is a scripting language originally created by Rasmus Lerdorf in 1994 for the creation of simple dynamic websites[1]. Since then, the language evolved into a full-featured object oriented language packed with a large number of standard libraries and dynamic features. Equipped with all these features, PHP makes it easy to get the job done, which resulted in a broad adoption. A research of W3Techs at service-side language usage among almost one million popular websites showed that in July 2013, 80.5% of the websites of which they knew the programming language was using PHP[2].

1.2 Dynamic Language Features

As most scripting languages, PHP has a number of dynamic language features. Dynamic language features let you delegate the precise behavior of your code at runtime. For example, instead of explicitly calling a certain function in the code, you can make use of dynamic invocation, where you call a method like `call_user_func`[3] with the function name as a argument in the form of a string. This way, the function to be called can be determined at runtime, instead of upfront. Listing 1.1 shows a simple example, where `call_user_func` is used reduce the number of lines of code. Instead of calling each function explicitly, the function names are synthesized as strings and called using `call_user_func`. Another example of dynamic invocation is shown in listing 1.2. This snippet is taken from WordPress and is used to delegate a function call to all installed filters, where a filter is a type of plugin. Here the function names are not known upfront due to the extensibility using plugins and therefore dynamic invocation is used. Another dynamic feature is `eval`. `eval`[4] lets you execute any string containing PHP source code.

```
foreach ( array('single', 'category', 'page', 'day',
                'month', 'year', 'home') as $type ) {
  $func = 'is_' . $type;
  if ( call_user_func($func) ) {
    $user_ts_type = $type;
    break;
  }
}
```

LISTING 1.1: Static usage example of `call_user_func`

```
foreach( (array) current($wp_filter['all']) as $the_ )
  if ( !is_null($the_['function']) )
    call_user_func_array($the_['function'], $args);
```

LISTING 1.2: Plugin usage example of `call_user_func_array`

1.3 Problem Statement

Dynamic features like dynamic invocation and eval can make PHP programs very flexible by determining which code should be executed at runtime, however it comes at the cost of code which is impossible to precisely inspect using static analysis. The main reason why this analysis is difficult is that, in case of dynamic invocation, by looking at the code it is not directly clear which function call will be invoked. To determine the actual function call you should know the value that is passed to the function that facilitates the dynamic invocation, for example the first argument of a `call_user_func` call. If this argument is a scalar, the behavior is trivial but the problem lies in the fact that this argument can be a variable and this variable can be set in any possible way, including via other dynamic features which are also hard to analyze because of the same reasons. Looking at listing 1.2, to know what functions will be called here would mean that you have to determine all the values of the array `$wp_filter['all']` which can become very complex. Next to that, the value can originate from a database or user input in which case the value cannot be determined by looking at the code. These problems for dynamic invocation are similar to the problems by having eval.

1.4 Motivation 1: Static Analysis

Because PHP is widely used, there is always a demand for better development tools. The key to creating a good IDE is besides general usability, the ability to support developers to write clean and correct software. Ways of giving that support are for example smart refactoring tools and possible bug detection. To be able to create features like that the PHP source code should be analyzed and interpreted in a certain degree to understand what it does. Here, static analysis comes into play. By making source code less dynamic, static analysis of this source code becomes more accurate. An example of a type of static analysis which is useful for an IDE is type inference. PHP is a dynamically typed language, which means that types are not explicitly stated in the source code. The type of a variable is determined by the value it is assigned. The process of determining the types of variables in a dynamically typed language using data-flow analysis is called type inference. When function calls are explicitly stated in the code, instead of variables passed into dynamic invocation functions, data-flow analysis becomes easier because the semantics of dynamic invocation functions do not have to be me known to determine the execution path. The data-flow analysis can also become more accurate because static code shows explicit execution paths instead of keeping them implicit which would be the case with dynamic invocation and therefore possibly undeterminable because of arguments that are passed into dynamic invocation functions that can originate from outside of the code base. More accurate data-flow analysis results into more accurate type inference which can be used to create a better type checker for PHP.

A second use case of static analysis for PHP is for security audits. Since most PHP code runs on web servers which are publicly reachable, security holes could have big consequences, for example user data being compromised. Next to manually inspecting source code to find these type of bugs, tools can be written which leverage static analysis techniques like data-flow analysis to detect security holes, such as possibilities for SQL injection by determining which variables are tainted[5]. As we have explained earlier, when the PHP source code contains less dynamic features static analysis becomes more accurate and will result in more accurate security audits which will result in less security holes.

1.5 Motivation 2: Performance

An other motivation for reducing the usage of dynamic features is possible performance improvements. Besides that dynamic invocation and eval can be slower than explicit invocation due to its indirectness, you call a function to call a function, is it possible that

PHP code with less dynamic features can better benefit from static analysis techniques that can be used to optimize code compilation[6].

1.6 Proposed Solution

In this thesis we will show a hybrid analysis technique to replace dynamic invocation and eval from PHP source with static source code.

The first part is a naive static approach. Static use cases of dynamic invocation methods and eval are being detected using straightforward pattern matching and will be replaced by the their explicit counterpart. A static use case is where a constant string is be passed to eval of one of the dynamic invocation methods, for example `eval("echo 'foo';")`. This call to the dynamic feature eval will immediately be replaced by `echo 'foo';`.

The more dynamically used occurrences of dynamic invocation and eval, where the arguments consists of variables, are replaced using information gathered by analyzing execution traces in the second part of the process. This dynamic analysis approach is because of the uncertainties that are related to analyzing dynamic features. These uncertainties come from the fact that values that are passed to dynamic invocation and eval are just strings and these strings could derive from everywhere within the source code, but also from other sources such as databases, command line parameters or HTTP's POST and GET variables. If such a string is originated from outside of the source code, it will not be possible to determine the value of this string using static analysis. To be able to capture all the values independent from their origin, we chose to observe them during execution.

At first the program is executed in an instrumented environment to be able to collect all the function calls including their parameters made during execution. Then we determine what functions signatures are passed to the dynamic invocation occurrences and what pieces of script are passed to eval. These usage patterns are then being incorporated directly into the code. If for example for a certain occurrence of `call_user_func($funcName)` two values (function names) for `$funcName` have been captured, this occurrence could be replaced by two if-statements which each have a test for one value of `$funcName` and execute the corresponding function call. The possible problems with using this dynamic approach is that the accuracy of your results depend on the code coverage you can achieve during execution and the representativeness of the execution for overall execution. This is because you can only capture the values you observe during execution.

The technique of using dynamic traces to study the usage of dynamic features is based on similar research into eliminating the dynamic language feature eval from code in the languages JavaScript[7, 8]. However they were able to look at already installed external systems because the client-side JavaScript of websites runs in the browser and is therefore available for inspection. PHP runs on the server, so we first have to install the software package we want to analyze to be able to instrument the execution.

For Ruby, another dynamic scripting language used on the server-side just like PHP, dynamic feature profiling was done by Furr[9]. Their approach is very similar as ours, the major difference is the language. What the added value of doing similar research for PHP is is its broad adaption. There is a lot more code written in PHP and because of the low barrier to start with PHP development, the quality of PHP code can vary a lot. All this code could benefit from better code analysis tools. Next to the different language Furr's research is focused on sound static analysis, whereas we aim for transformed code that is suitable for unsound static analysis. Unsound static analysis is when you for example look at security flaws but do not capture them all because the transformed code is an under-approximation of code. The power of unsound analysis is that it is faster to do since you do not look at every possible execution path, which shrinks the solution space. You also do not suffer from a lot of false positives in security analysis, which would derive from undecidable execution paths that have to be marked as insecure when you want to be sound. Having many false positives could result in a situation where the person reviewing the analysis results is not able to see the wood for the trees.

These earlier researches into JavaScript and Ruby have shown that in most cases dynamic code behavior is predictable and can be transformed to a static variant. This gives us confidence, similar findings could be done in PHP.

1.7 Hypotheses

The accuracy of static analysis on PHP code can be improved by reducing number of occurrences of dynamic invocation and eval using dynamic analysis.

The performance of PHP code can be improved by reducing number of occurrences of dynamic invocation and eval using dynamic analysis.

1.8 Outline

This thesis will first start with describing related work (chapter 2) in this research area. What follows is the research into the usage of dynamic features in a corpus of 19 PHP software packages and a more in-dept look at WordPress (chapter 3). Next we describe how we implemented our solution (chapter 4). The following two chapters 5 and 6 show how we evaluate our two hypotheses. After that we end with the conclusions and future work (chapter 7).

Chapter 2

Related Work

In this chapter we will discuss some of the other research done in this field and how it relates to our work.

2.1 PHP Feature Usage

In a paper by Hills et al.[10] a broad research is done on the usage of different features of PHP in a corpus of 19 PHP systems, including the 12 of the most popular used PHP projects. The goal was to investigate which language features should be dealt with in order to be able to develop static analysis tools for PHP. During the research they stumbled upon multiple dynamic language features that impose a problem for static analysis. With a few smart lightweight techniques they were able to resolve some of the dynamic references, for example dynamic includes and variable variables, to improve the ability for static analysis. They also looked at the amount of dynamic invocation usage in each of the analyzed PHP system. Per system they counted in how many files dynamic invocation occurred directly or indirectly via dynamic includes. What they found is that the number of files in a project that contain at least one occurrence of dynamic invocation ranges between 2 and 92. This indicates that for certain systems the number of dynamic invocation is significant. Since dynamic invocation is significant we found it worthwhile to further investigate this dynamic feature.

2.2 Dynamic Analysis

Richard et al.[7] performed a study which does an in-depth analysis of the usage of eval in the JavaScript code of large number of websites. It was performed by gathering traces using a modified web browser. Their conclusion was that there are a number of popular use cases of eval which could be replaced with safer strategies, however there are still cases where the use of eval is inevitable. A similar runtime analysis of PHP code is possible for determining the different use cases of dynamic invocation and eval.

Continuing on the previous study, Meawad et al.[8] describe how they were able to replace a lot of eval occurrences in JavaScript programs by analyzing every string that was passed to eval, extract usage patterns and replacing the eval with substitution code including a fall back mechanism to eval itself. The way they were able to model the usage of eval at certain places by analyzing the dynamic traces should be possible to implement in PHP as well. However, since PHP is a server-side scripting language, our traces should be gathered at the server-side on PHP installations we have control over ourselves. This makes it more involved to capture accurate dynamic behavior of PHP systems, but the basic idea of leveraging execution traces to investigate dynamic features is an approach we will take as well.

In 2009 Furr et al.[9] tried to statically type source code written in the dynamic scripting language Ruby. To be able to infer the types using static analysis, they first replace dynamic pieces of code with a static equivalent. This code transformation is guided by dynamic tracing how these dynamic features are being used. This data is collected through profiling the application code. This is a similar approach as what is done by Meawad et al.[8] The research showed that most occurrences of dynamic features were transformable to static code. Ruby being a server-side scripting language mostly used for the Web just like PHP brings hope that equally impressive results can be achieved. Instead of type inference, our goal will be better unsound analysis and possible performance improvements.

2.3 Static Analysis of PHP

Jovanovic et al.[5] created a tool called Pixy which uses data-flow analysis to detect security breaches in PHP source code in the form of SQL injection possibilities. The data-flow analysis is done purely using static source code analysis, which give them the advantage of inspecting all the code paths without execution. However, they do not support automatic file inclusion because of PHP's dynamic nature. We think that by making the source less dynamic by transforming the source code using information gathered by dynamic analysis, would further improve the accuracy of this type of static analysis. After the source code is transformed by our tool, tools like Pixy should get more accurate results.

At Facebook, they have a large PHP code base. In order to get better performance out of this interpreted dynamic language they decided to static compile the PHP code, with their own compiler called HipHop[11]. The ahead-of-time compilation process consists of a number of phases including, program analysis and type inference which is all done in a static manner. Although HipHop has shown a great performance improvement, it enforces some restrictions to the programmers because of the lack of a few dynamic features such as dynamic code evaluation (eval). Facebook's main goal is to increase the performance of their code and they are able to restrict their developers to not use certain features like eval. Even though we share the goal of performance improvements, we also want to be able to handle systems using these dynamic features. Next to that we have a second goal that is to modify the code itself to make it more suitable for further analysis.

Another earlier attempt to compile PHP to machine code was Biggar's PHC[12]. Just like Hiphop, PHC does a number of static optimizations such as constant-folding to get better performance. In order to support `eval` they have coupled the PHP runtime in their compiler and compiled code. This of course, does not reduce the performance penalty of eval. Another thing, just as we mentioned for Facebook's HipHop compiler, our additional goal is to prepare the PHP source for further analysis, not converting it to C++ code. We want to analyze and possibly optimize dynamic feature usage in PHP code.

Chapter 3

Usage of Dynamic Features

This chapter describes the research into the behavior of the dynamic features we are interested in to understand how it fits our proposed solution.

3.1 Dynamic Invocation and Eval

In PHP, there are two functions which can be used for dynamic invocation. To get everyone onto the same page, we will briefly explain their syntax and semantics.

`call_user_func( $callback [, $parameter [, $... ]] )` The first parameter is the callback and all the following parameters are passed to the callback. `call_user_func` is a variadic function with one mandatory parameter. This means that it can handle an arbitrary number of parameters with a minimum of one. The return value of the callback is returned.

`call_user_func_array( $callback , array $paramArr )` The first parameter is the callback and the second parameter is an array of all the parameters that are passed to the callback. The return value of the callback is returned.

The callback is the function or method that should be called. A callback can have a number of forms. When it is a function, the callback is just a string of the name of the function (`"functionName"`). For non-static method calls, the callback is an array with the object as first item and the method name as the second item in the form of a string (`array($object, "methodName")`). The third form is for static method calls, there the callback is an array with the first item being a string containing the name of the class and the second item also a string containing the method name (`array("className",` `"methodName")`). Since PHP 5.3 the callback can also be a closure, however, since this use case is rarely used, we do not that take them into account in our research.

Furthermore we also look at `eval`. Here follows a short description.

`eval ( string $code )` The parameter is a string containing the code that should be executed. Eval can return a value if the code contains a `return` statement. At the moment we do not support the return value of eval.

3.2 Occurrences in Corpus

By counting the calls of the two primary functions for dynamic invocation `call_user_func` and `call_user_func_array` in the initial corpus that was put together by Hills et al.[10] we reached a total of 931 ocurrences. To get an idea of how they are generally used we manually inspected them. When looking at each of the occurrences, all of them can be put in one of the following categories:

Plugin/Module/Hook infrastructure. A number of the software packages in the corpus are frameworks or content management systems, which are meant to function as the base of a software system and are designed to be able to easily be extended. To achieve this extensibility, the software packages have all types of plugin systems and to handle plugins of which the class and method names are unknown upfront, dynamic invocation is used. An example of this can be seen in listing 1.2.

Calling callback functions. There are scenarios where you would want functions that can accept callback functions as arguments which should then be called in certain circumstances in the code. These arguments are also known as anonymous functions or closures. To be able to call a such an anonymous function you need to use dynamic invocation.

Unit test frameworks. When automatically running test suites, unit test frameworks like PHPUnit use dynamic invocation to call test cases and their setup and teardown methods.

Proxied method calls in magic methods. PHP has the notion of magic methods. These are class methods with special names which are implicitly called when certain actions are performed on that class. One magic method which can be implemented in a class is `__call`. This method is invoked when an inaccessible method of an object instance is being called, even if the objects class does not have an implementation of this method at all. Some of the software packages in the corpus use these types of magic methods and use dynamic invocation to forward the method call to a another class.

Passing array of arguments. One trick for which `call_user_func_array` is used is to pass the elements of an array as arguments in a function call because second parameter of `call_user_func_array` is an array of arguments.

Configuration. Similar to the case of plugin infrastructures, there are configuration components in software packages to make them more flexible. We found that there are use cases in which certain class names are set in these configuration settings, for example to determining which database driver to use. Dynamic invocation is then used to call the configured classes.

Code shortcuts. In order to write the least code possible and not repeat yourself, there are cases where `call_user_func` is used in loops that just iterate over static strings and where these strings are used as arguments of `call_user_func`. These loops could easily be unrolled and `call_user_func` could be avoided, but this would make the code more verbose and less maintainable. An example of this can be seen in listing 1.1

3.2.1 Threats to Validity

Even though the variety in terms of application domain within the corpus is high, there can always be use cases of dynamic invocation that are not occurring in this corpus. This makes this categorization not exhaustive and causes a threat to external validity. Furthermore, the classification is done by inspecting the call sites and their immediate contexts, where the context was limited to the function or method in which a call occurs so their exact behavior could be misunderstand. Next to that this type of classification is subject to interpretation. Some cases could be put in more than one category. These factors have a negative impact on the reliability of this categorization. A more objective categorization could be achieved by defining strict criteria on which every occurrence would be rated and use these ratings to segment them into categories. However, the goal of this part of the research was to give an overview of typical use cases of dynamic invocation and to get an idea of how these use cases will fit our proposed solution of using dynamic traces to replace the occurrences with explicit function calls.

3.2.2 Conclusion

When inspecting the occurrences of dynamic invocation in a corpus of 19 PHP projects, we observed that most use cases of dynamic invocation that we inspected are related to plugins and configuration. This gives us an angle for further investigating the usage of dynamic invocation.

3.3 Case Study: Execution Traces of WordPress

Manually analyzing occurrences of dynamic invocation shows us what type of use cases there are, but gives us no insight into how dynamic these occurrences are used. To actually look at how dynamic invocation behaves in a real world scenario, we have to install an application, instrument the installation and execution the application.

For this more in-dept analysis of dynamic invocation in a widely used application, we chose to take a look at WordPress[13]. WordPress is a content management system which can be used to create websites. According to WordPress co-founder Matt Mullenweg in July 2013, 18.9% of the Web is powered by WordPress[14]. Because of this large installed base, it seems a good candidate for a case study.

In addition to inspecting the source code to see how dynamic invocation and `eval` is being used, it is possible to instrument the runtime environment to see what arguments are explicitly passed to the two main dynamic invocation related methods `call_user_func` and `call_user_func_array` as well as `eval`. For this research the PHP extension Xdebug[15] is used. In order to execute the code and simulate real world usage, multiple requests are being triggered by a functional test written for CasperJS[16].

To get an idea of how dynamic `call_user_func` and `call_user_func_array` are being used in a WordPress 3.5.2 installation table 3.1 and 3.2 give an overview of the number of unique function signatures that are dynamically being invoked. The definition of function signature in this context is the method or function that is being called combined with the number of arguments. The number of arguments is included since PHP supports variadic functions and `call_user_func_array` uses an array for passing in the arguments, which allows for a dynamic number of arguments per call site.

The tables show the variety of function signatures per covered dynamic invocation call site for 5 different WordPress installations ranging from 0 plugins to 4 plugins installed. Table 3.3 show an ordered list of the plugins that are added to the installation to get the desired number of plugins. The reason we distinguish between the number of plugins installed is based on our earlier observation that a typical motivation for the

Location (<file>:<line number>)	Number of Plugins				
	0	1	2	3	4
wp-admin/includes/template.php:927	24	24	24	25	25
wp-admin/includes/class-wp-list-table.php:860	7	7	7	7	7
wp-includes/media.php:1230	2	2	2	2	2
wp-includes/category-template.php:674	2	1	1	1	1
wp-includes/comment-template.php:1334	1	1	1	1	1
wp-includes/functions.php:2035	1	1	1	1	1
wp-includes/class-http.php:216	1	1	1	1	1
wp-admin/custom-header.php:471	1	1	1	1	1
wp-includes/nav-menu-template.php:179	1	1	1	1	1
wp-includes/SimplePie/Parse/Date.php:602	0	2	2	2	2
wp-includes/class-simplepie.php:1338	0	1	1	1	1

Table 3.1: # of different signatures per `call_user_func` in WordPress

Location (<file>:<line number>)	Number of Plugins				
	0	1	2	3	4
wp-includes/plugin.php:406	100	135	156	207	207
wp-includes/plugin.php:173	63	64	68	79	79
wp-includes/plugin.php:487	14	20	22	24	24
wp-admin/includes/widgets.php:196	12	12	12	12	12
wp-includes/class-wp-walker.php:129	7	7	7	7	7
wp-includes/class-wp-walker.php:157	7	7	7	7	7
wp-includes/widgets.php:893	7	7	7	7	7
wp-includes/capabilities.php:926	2	2	2	2	2
wp-includes/capabilities.php:1289	2	2	2	2	2
wp-includes/post-template.php:964	1	1	1	1	1
wp-includes/nav-menu-template.php:478	1	1	1	1	1
wp-admin/includes/widgets.php:45	1	1	1	1	1
wp-admin/includes/template.php:149	1	1	1	1	1
wp-admin/includes/template.php:146	1	1	1	1	1
wp-includes/category-template.php:738	1	1	1	1	1
wp-includes/post-template.php:947	1	1	1	1	1
wp-includes/plugin.php:230	1	1	1	1	1
wp-includes/category-template.php:756	1	1	1	1	1
wp-includes/SimplePie/Registry.php:222	0	7	7	7	7
wp-admin/includes/dashboard.php:1049	0	4	4	4	4
wp-cron.php:87	0	1	1	1	1
wp-includes/SimplePie/Registry.php:215	0	1	1	1	1
wp-content/plugins/jetpack/class.jetpack-sync.php:62	0	0	1	1	1

Table 3.2: # of different signatures per `call_user_func_array` in WordPress

use of dynamic invocation is to facilitate a plugin architecture. Working with plugins means that you are going the work with classes and methods of which the names are unknown upfront. To see this observation in practice, we show the variety per number of plugins installed.

#	Plugin	Version
1	Contact Form 7	3.4.2
2	Jetpack by WordPress.com	1.6
3	WordPress SEO	1.4.13
4	WordPress Importer	0.6.1

Table 3.3: Plugins used

Looking at the table 3.1 and 3.2 it seems that there are use cases of dynamic invocation in WordPress that are not that dynamic at all. 6 out of the 11 covered `call_user_func` call sites and 11 out of 23 covered `call_user_func_array` call sites actually only get one unique function signature passed to them. This could indicate that these cases could be rewritten without the use of dynamic invocation, since that single function signature could just be called explicitly.

When we inspect how the different number of plugins per installation influence the variety of function signatures, table 3.1 does not show any significant influence. Table 3.2 however shows for the top 3 varying call sites of `call_user_func_array` a clear increase of variety when the number of plugins increases. These call sites are unsurprisingly in a file called `plugin.php`. This observation of a correlation between plugin usage and dynamic invocation variety confirms our earlier observation, that dynamic invocation is used for plugin architectures.

3.3.1 Threats to Validity

Since our data about dynamic invocation is gathered through dynamic analysis, the program has to be executed. Dynamic analysis is as good as the coverage of its execution. The test case we created for executing WordPress is handcrafted and covers 18 out of 47 (38.3%) `call_user_func_array` occurrences and 9 out of 42 (21.4%) `call_user_func` occurrences. 5 occurrences of `call_user_func_array` where excluded from these numbers, since they are located in deprecated functions and our installation including all 4 plugins do not contain explicit calls to these deprecated functions. Not covering the majority of the occurrences of dynamic invocation can decrease our validity.

During the analysis of WordPress, we observed that the plugins are a typical use case for dynamic invocation and that when the number of plugins increases, the variety of function signatures passed to dynamic invocation functions also increases. This relation is causal, however we only observed this in WordPress which could be a threat to external validity. Another observation we did was the relative large amounts of occurrences that receive just 1 function signature. This could indicate that these are actually static use cases, but given the fact that this observation is based on dynamic tracing which was limited to the coverage of the functional test cases, these occurrences could possibly behave more dynamic than we were able to see. A better coverage could have given us more valid results. Also an additional independent variable next to the number of plugins could possibly give us more insight on which factors determine the dynamics of dynamic invocation.

3.3.2 Conclusion

This analysis of WordPress shows us that plugin infrastructures are an important influence on the behavior of dynamic invocation. This gives us some level of confidence that when the configuration of an application is stabilized, occurrences of dynamic invocation are used in a static manner, meaning that they always call a fixed set of function and methods.

3.4 Dynamic Invocation Stability Theory

Since we have seen that a lot of the usage of dynamic invocation is related to plugins and configuration, we hypothesize that this dynamic behavior will be static when a system is configured and running. The following stages of a PHP system can be distinguished with respect to the stability of dynamic invocation usage patterns: *configuration* and *deployed*. See figure 3.1.

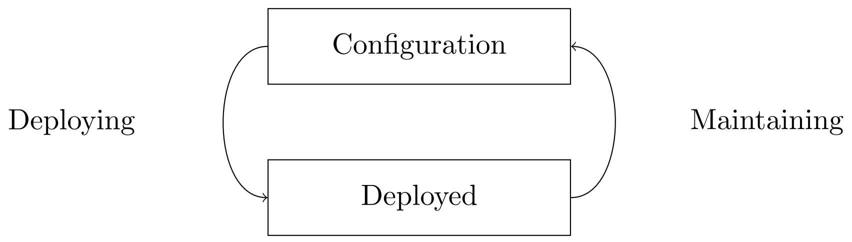

Figure 3.1: Phases of PHP system

In the configuration phase, plugins are installed and configured, this in contrast to the deployed phase, where we assume that the system and its environment is stable.

Because we would like to replace the occurrences of `call_user_func` as well as the occurrences of `call_user_func_array` with inline explicit function calls, we need to analyze the system in its stable form to ensure consistent behavior of the dynamic invocation occurrences. The analyzing process will consist of execution trace gathering, preferably in the live environment to get the most accurate results. Our source code transformations will then be based on the current state of this stable form. Figure 3.2 shows how these phases relate to each other.

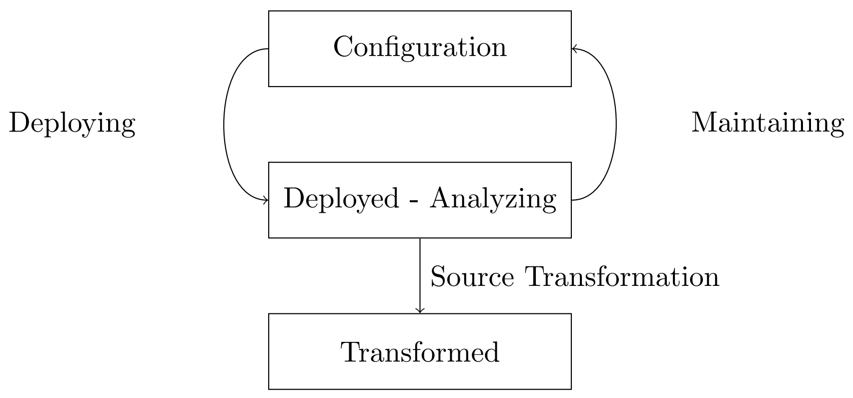

Figure 3.2: Phases of PHP system (extended)

3.4.1 Threats to Validity

This theory depends on the assumption that plugins and configuration are the main contributors to the dynamic behavior of dynamic invocation and therefore a stable state of the application exists. However, scenarios could exist where external factors such as user input influence the variety of function signatures passed into `call_user_func` and `call_user_func_array`. In that case there is no stable state of the application.

3.4.2 Conclusion

Since the plugins installed and the configuration of a system tend to stabilize when a system is deployed and since we assume that the behavior of dynamic invocation becomes more static when the system is stable, we propose that to have the most accurate trace information it is best to trace a deployed system. Also the deployed environment is where the system lives and therefore traces gathered from this environment will give the most accurate representation of how the system behaves. The assumption that a stable form of the system exists gives us the confidence that we are able to of safely replace dynamic invocation with explicit invocation.

Chapter 4

Implementation

In this chapter we will describe which requirements and design decisions lead to the tool we created, how the tool was constructed and how all the underlying algorithms work.

4.1 Design

Since dynamic invocation is hard to analyze using static analysis techniques, because the values of the arguments have to be determined, the goal is to construct a tool that replaces the occurrences of `call_user_func` as well as the occurrences of `call_user_func_array` with explicit function calls. In addition to reduce dynamic invocation, the tool will also inline pieces of code that are passed into `eval`. The process will be based on dynamic analysis of the program combined with transformation on the source code.

4.1.1 Requirements

Functional Requirements

1. **Valid Output.** The tool should replace the occurrences of dynamic invocation and `eval` from the given PHP system with semantically equivalent PHP code.

   Rationale: The outputted PHP system should be suitable either as a replacement for the original system in its execution environment or as input for further source code analysis. Therefore it is important that the transformed source code works exactly as the original code.

2. **Automation.** The tool should be able to be automated.

Rationale: When you could automate the interaction with the tool, it could be integrated into other tools like IDEs and continuous integration systems.

Non-Functional Requirements

1. **Modularity.** The tool should have a modular architecture.

   Rationale: This ensures better separation of concerns. This separation makes it easier to extend the systems functionality, for example replacing the tracing module with a new, faster implementation or adding transformation algorithms for other types of dynamic features. Having a modular setup ensures that for each of these examples the whole system does not have to be altered, but only the related module.

2. **Fast.** The tool should use the least number of resources as possible.

   Rationale: Because this tool needs to run next to other development tools like IDEs, we should try to keep the setup as efficient as possible.

4.1.2 Design Decisions

1. **Split-up Tracing and Transformation.** In the tool we decided to split-up the tracing phase of the process from the analysis and transformation phase.

   Rationale: We did this to have a modular setup. Alternatively, we could integrated it all into one program, but this would increase the complexity of this one program and would not allow the user to execution both phases of the process separately. Next to that, it will me easier for different parts of the program to be altered or replaced.

   Related requirement: NFR1

2. **Xdebug for Tracing.** We decided to use the open source PHP extension XDEBUG for instrumenting the PHP execution environment to gather trace information.

   Rationale: Alternatively, we could have altered the PHP interpreter ourselves since it is open source, but we chose to go for an already existing and proven alternative. Also because it is not strongly coupled to our system it could easily be replaced if necessary.

   Related requirement: NFR1

3. **Lighttpd as Server.** We decided to use LIGHTTPD combined with PHP and MYSQL in our server setup to run the PHP code.

Rationale: We chose LIGHTTPD for its small footprint, alternatively, we could have chosen APACHE or NGINX, since these are more widely used, but this choice would not impact the quality of the tool.

Related requirement: NFR2

4. **AWK to convert output to CSV.** We decided to use AWK to filter and convert the output of XDEBUG on-the-fly.

   Rationale: We chose AWK for its small footprint and the ability to handle the file stream outputted by XDEBUG directly to limit disk usage.

   Related requirement: NFR2

5. **Functional Testing with CasperJS.** We decided to use the CASPERJS to run the functional tests that execute the PHP application.

   Rationale: CASPERJS is an open source test framework which uses an headless browser. This makes it easy to run from a command line environment and therefore suitable for automation.

   Related requirement: FR2

6. **Rascal for Analysis and Transformation.** We decided to use the RASCAL language for analysis of the traces and transformation of the source code.

   Rationale: There already was a parser for PHP that generated RASCAL ASTs and RASCAL has powerful features to manipulate an AST which made it easy to transform the PHP source code.

   Related requirement: FR1

4.1.3 Traceability

During this chapter we will refer to these requirements and design decisions using a prefix and their number. For the functional requirements we have the prefix *FR*, the non-functional requirements will use *NFR* and we will use the prefix *DD* for referring to the design decisions.

4.2 Execution Trace Gathering

In order to see how the occurrences of the dynamic features that we are interested in are being called during runtime, the PHP extension XDEBUG (DD2) is being leveraged. This open source extension brings extra debug utilities such as clearer stack traces and better timing methods to PHP but also the ability to get logs of all the function calls that were executed including the used arguments. This logging feature is used to gather all the arguments that are being passed into `call_user_func`, `call_user_func_array` and `eval` without modifying PHP ourselves.

To generate logs, the code has to be executed. One way to do this is by running the unit tests that are available for the software project in question. Even though this will generate logs with possibly a good code coverage, unit tests are not always representative for real world execution. Therefore we did not chose unit test but installed the software on a lightweight web server setup consisting of LIGHTTPD (DD3), PHP and MYSQL. Functional tests written for the CASPERJS (DD5) framework will generate the HTTP requests to the installed software by simulating real user behavior. These tests have to be specially crafted for the PHP system under investigation. One disadvantage of using unit tests or functional test is that it is hard to get 100% coverage.

The logs generated by XDEBUG can become very large because they contain all function calls made during execution including their arguments. We filter the output of XDEBUG and convert it into comma-separated value (CSV) (DD1) format using AWK (FR2) so that the traces are suited for easy analysis in the second part of the tool: code transformation in Rascal. Figure 4.1 gives an overview of the whole system we used to gather execution traces.

4.3 Code Transformation

4.3.1 Rascal

RASCAL is a programming language created at CWI. The language's main purpose is to facilitate metaprogramming[17]. It does this by having integrated support for generating parsers and having powerful pattern matching and tree traversal capabilities. Part of our tool is written in RASCAL (DD6) and is build upon earlier work of Hills[10] that allows us to parse PHP source code to abstract syntax trees (AST). The AST structure can than be easily inspected and altered using language features natively available in RASCAL. The resulting AST can than be converted back to PHP source code (FR1).

4.3.2 Algorithm

The Rascal program first starts with converting each PHP file into an AST. This is done using the PHP analysis framework which is provided by Hills. Now we have the

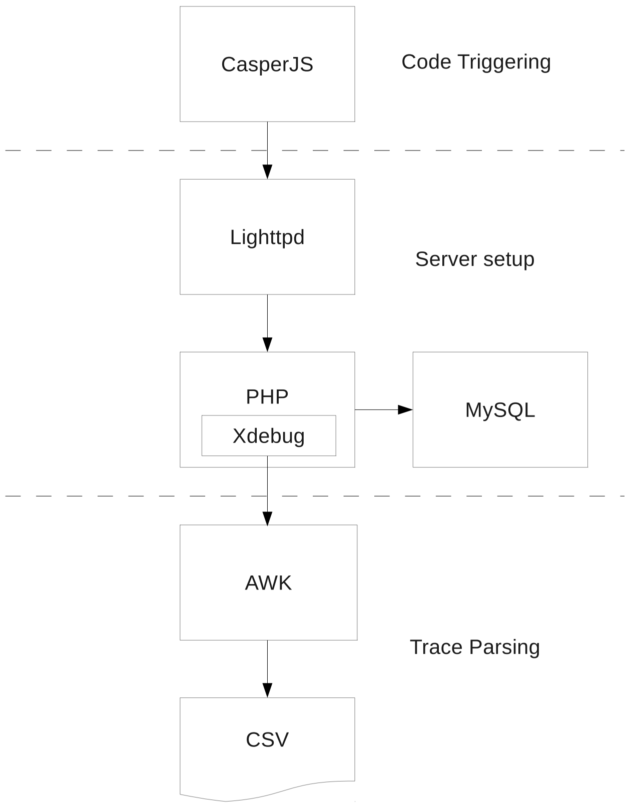

FIGURE 4.1: Tracing Architecture

ASTs we start with a naive form of static analysis. There can be occurrences of dynamic invocation and eval which are not dynamic at all. For instance, when a static string is passed as the first argument of `call_user_func` like this `call_user_func("func", $arg)`. This call could be replaced by `func($arg)`. Even though these cases are rare, they are easy to detect and replace and it is possible that other static analysis techniques like constant propagation will result into such occurrences being present. By running a pattern matching algorithm against the AST of the source code, these static use cases are detected. The detected code fragments are then replaced the explicit method calls in the case of dynamic invocation and inline code for eval.

The remaining occurrences of dynamic invocation and `eval` will be handled in the second phase. In RASCAL the CSV file containing all the logs of `call_user_func`, `call_user_func_array` and `eval` is parsed so that we only have unique function signatures for each dynamic invocation call site and unique arguments per `eval` call site. We parse the serialized callback argument we got from XDEBUG for dynamic invocation function calls into one of the three callback types: function call, non-static method call and static method call. For `call_user_func_array` we also look at the number of array elements of the second argument. The combination of the callback and number of arguments make up the function signature. Using the known function signatures for the `call_user_` `func` and `call_user_func_array`, static code is generated which does these function invocation directly. `eval` is a special case, the script string that is passed to `eval` is directly inlined within the source code. Algorithm 1 gives a general overview of the main transformation algorithm.

Algorithm 1 General algorithm of code transformation

```
for all $o$ in $occurrences$ such that $o$ has traces do
  $traces \leftarrow$ traces of $o$
  $context \leftarrow$ surrounding statement of $o$
  $statements \leftarrow \emptyset$
  for all $t$ in $traces$ do
    $s \leftarrow$ generate if-statement for $t$ with $context$
    add $s$ to $statements$
  end for
  if has to be sound then
    add $context$ as else case to $statements$
  end if
  replace $context$ with $statements$
end for
```

The following algorithm (2) described how an if-statement is generated for a trace of dynamic invocation.

Algorithm 2 If-statement generation for a dynamic invocation trace

```
$callback \leftarrow$ callback variable
$arguments \leftarrow$ list of arguments to pass to callback
$conditions \leftarrow \emptyset$
if $trace$ is function call then
  $conditions \leftarrow$ ($callback$ == "functionName")
  $body \leftarrow context$ with occurrence replaced by functionName($arguments$...)
  return if-statement with $conditions$ and $body$
end if
add ($callback$ is array) to $condictions$
if $trace$ is non-static method call then
  add ($callback$[0] is object) to $condictions$
  $explicitCall \leftarrow$ callback[0]->methodName($arguments$...)
else if $trace$ is static method call then
  add ($callback$[0] == "className") to $condictions$
  $explicitCall \leftarrow$ className::methodName($arguments$...)
end if
add ($callback$[1] == "methodName") to $condictions$
$body \leftarrow context$ with occurrence replaced by $explicitCall$
return if-statement with $conditions$ and $body$
```

We will now demonstrate the workings of the algorithm with a simple piece of code that contains an occurrence of `call_user_func`. First we will show you the original code (listing 4.1), followed by two transformed versions (listing 4.2 and 4.3).

```php
<?php
function func1() {
    return "foo";
}
function func2() {
    return "bar";
}

function func3($funcName) {
    return call_user_func($funcName);
}

func3("func1");
func3("func2");
?>
```

Listing 4.1: Simple example of `call_user_func`

Given the code fragment shown in listing 4.1, our execution log will tell us that the following two string values were passed as the callback of this call site of `call_user_func`: `"func1"` and `"func2"`. Using this information one of the following two pieces source code (listing 4.2 and 4.3) will be generated:

```php
<?php
function func1() {
    return "foo";
}
function func2() {
    return "bar";
}

function func3($funcName) {
    if ($funcName == "func1" && function_exists("func1")) {
        return func1();
    } else if ($funcName == "func2" && function_exists("func2")) {
        return func2();
    } else {
        return call_user_func($funcName);
    }
}

func3("func1");
func3("func2");
?>
```

Listing 4.2: Resulting code of `call_user_func` (sound)

```php
<?php
function func1() {
    return "foo";
}
function func2() {
    return "bar";
}

function func3($funcName) {
    if ($funcName == "func1" && function_exists("func1")) {
        return func1();
    } else if ($funcName == "func2" && function_exists("func2")) {
        return func2();
    }
}

func3("func1");
func3("func2");
?>
```

LISTING 4.3: Resulting code of `call_user_func` (unsound)

The resulting code (listing 4.2 and listing 4.3) is constructed using the information gathered by looking at execution traces to make the possible execution paths more explicit. As you can see in listing 4.2, the code still contains the call to `call_user_func`, this in case that there are values of `$funcName` that are not seen during the execution. This guarantees that the transformed program can cover all possible executions paths of the original program and makes it suitable for execution. However, this `else` case could be removed when you are certain that all possible values are covered or when there is a need for unsound static analysis, listing 4.3 show an example how the source code will look like for that case.

Another interesting fact to note is that the surrounding statement of the `call_user_func` method is also copied to each `if` case, in this example the `return` statement, to assure that the semantics of the code stay the same.

Now for a more complicated example, we will show the transformation of `call_user_func_array` call:

```
<?php
class A {
    public function nonStaticMethod($param) { }
    public static function staticMethod($param1, $param2) { }
}

$object = new A();

$callback = array($object, "nonStaticMethod");
$args = array("arg");
call_user_func_array($callback, $args);

$callback = array("A", "staticMethod");
$args = array("arg1", "arg2");
call_user_func_array($callback, $args);
?>
```

Listing 4.4: Example of `call_user_func_array`

Execution of this code will result in two execution traces, one for each occurrence. The first trace be the non-static method callback and one argument in the array and the second trace the static method with two arguments. The sound version of the transformed code looks like this (listing 4.5):

```php
<?php
class A {
    public function nonStaticMethod($param) {}
    public static function staticMethod($param1,$param2) {}
}

$object = new A();
$callback = array($object,"nonStaticMethod");
$args = array("arg");
if( is_array($callback) && sizeof($callback) > 1 &&
  is_object($callback[0]) && $callback[1] == "nonStaticMethod" &&
  method_exists($callback[0],"nonStaticMethod")  &&
  is_array($args) && sizeof($args) == 1) {

    $callback[0]->nonStaticMethod($args[0]);
} else {
    call_user_func_array($callback, $args);
}

$callback = array("A", "staticMethod");
$args = array("arg1", "arg2");
if( is_array($callback) && sizeof($callback) > 1  &&
  $callback[0] == "A" && $callback[1] == "staticMethod" &&
  method_exists("A","staticMethod") &&
  is_array($args) && sizeof($args) == 2) {

    A::staticMethod($args[0], $args[1]);
} else {
    call_user_func_array($callback,$args);
}
?>
```

Listing 4.5: Example of `call_user_func_array`

As you can see, the conditions of the if-statements in listing 4.5 are quite elaborate. This extensive checking is to make sure that the resulting code does not generate any errors.

4.4 Ensuring the Correctness

4.4.1 Preconditions

To ensure that the transformed code acts exactly the same as the original code and does not introduce any errors, all inserted function and method calls are preceded with a number of checks on preconditions. As we have explained in the beginning of chapter

3, there are three types of callbacks which can be passed to `call_user_func` and `call_user_func_array` that we are interested in. For each of these types the preconditions are slightly different as we have shown in algorithm 2.

The first type of callback is a function name. Here the callback is a string of the name of the function, as seen in listing 4.1. In the transformed code, first we check if the function should be called by checking the value of the callback. Next to that, we make sure that function exists to be sure that the explicit function call will not result in a fatal error (listing 4.3).

The second and third type of callback are respectively non-static and static method calls. The callback for these types is an array consisting of two elements. For non-static method calls the first element is an object and for static method calls the first element is the class name as a string. The second element for both types is the method name as a string. To be sure the correct explicit method call is made in the transformed code, we first check if the callback is an array with at least two elements. Then we check for non-static method calls if the first element is an object. For static method calls the first element should be the name of class of the explicit call. The second element should match the name of the method. Then we check if the method exists.

These preconditions are all checked for the `call_user_func` and `call_user_func_array` replacement code. However, as you can see in listing 4.5, for `call_user_func_array` replacements to be safe additional checks are done on the argument array. First the array variable is checked if it indeed is an array and after that the number of elements is validated, this number should correspond with the number of arguments in the explicit function or method call to prevent out of bound access of the array.

Using these preconditions we can make sure that the transformed code acts as the original and does not trigger any errors that are not triggered by the original code.

4.4.2 Context Preservation

Since the dynamic invocation methods are expressions, they can occur at the same places as any other expression can occur, for example as the value in an assignment. In our transformation we wrap the explicit function and method calls with conditional statements, but we cannot directly replace a dynamic invocation expression with these statements. To make sure that the call is made in the same context as the original dynamic invocation call we take statement directly surrounding the original expression and wrap it around the explicit calls in each of the bodies of the conditional statements.

An example of this can be seen in listing 4.3 where the surrounding return statement is preserved in all of the cases.

By preserving the context in which the dynamic invocation expression is occurring, we can guarantee that the semantics of the transformed code is the same as that of the original code.

Chapter 5

Evaluation of Static Analysis Improvements

In this chapter we will evaluate the improvements that are gained using are implemented tool considering further static analysis of a given processed PHP system.

5.1 Simple Graph Experiment

Our first motivation for reducing the number of dynamic features was to improve the opportunities for static analysis. Using a small example, we will show how eliminating dynamic invocation could achieve this.

When performing static analysis, the goal is to extract facts from source code. One well know type of static analysis is call graph analysis (CGA). A call graph is a graph which shows which units of a program call which other units. Some typical units are classes, files, modules and functions.

Imagine creating a call graph for function calls of the following code:

```
<?php
function A() {
    global $funcName;
    call_user_func($funcName)
}

function B() {
    C();
    D();
}

function C() { }

function D() { }
?>
```

Listing 5.1: CGA example with `call_user_func`

Assuming we are not able to statically resolve the global variable `$funcName`, function `A` could potentially call every other function or method. The resulting call graph would look as follows:

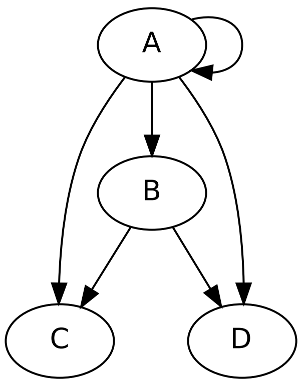

Figure 5.1: Call graph with `call_user_func`

The function `A` has an arrow to every other function. However, it could be that in reality function `A` is unlikely to call every single function or method. Now let say that we perform our dynamic tracing process on this piece of code and that we intercept one value of `$funcName` which is `"B"`. Utilizing this trace, the code can be transformed to the this:

```
<?php
function A() {
    global $funcName;
    if ($funcName == "B") {
        B();
    }
}

function B() {
    C();
    D();
}

function C() { }

function D() { }
?>
```

Listing 5.2: CGA example with `call_user_func`

The code generated is an under-approximation of the original code, since there could be possible execution paths in the original code, that are not possible in this modified version. Using this code will therefore result in unsound analysis. The code of listing 5.2 will result in a much cleaning call graph (see figure 5.2).

$$\begin{tikzcd}
 & A \arrow[d] & \\
 & B \arrow[dl] \arrow[dr] & \\
C & & D
\end{tikzcd}$$

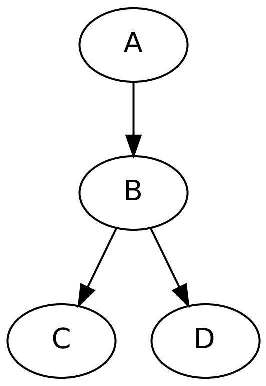

Figure 5.2: Call graph without `call_user_func`

This call graph gives a more clear view of the real behavior of the program in comparison of the previous graph, however we can not be sure if this represents all the possible calls.

5.1.1 Threats to Validity

We demonstrate for a small example program, how we remove false positives among the edges of the call graph. The resulted call graph however, is not necessarily more precise than the original one, since it is based on an under-approximation of the original code. It is possible that we only registered one value for the particular call site of `call_user_func` but in practice there are more values. This is the same problem as the problem with the coverage of the functional tests. The dynamic analysis is dependent on the completeness of the traced execution and the level to which it represents real world execution. As we described in the model, the deployed application could be considered stable and is the best execution environment to capture real world usage.

5.1.2 Conclusion

The transformed code gives us a cleaner ouput of our static analysis, but since it is unsound analysis, it is hard to determine if the accuracy really increases.

5.2 Call Graph Generation of WordPress

Next to the previous simplified example to show the implications of code transformations on the accuracy of call graph generation, we also looked at a real application. As we did early we chose WordPress as our benchmark. Using Doxygen, an open source tool to generate documentation from source code we are able to generate call graphs for every function and method in the code base. For a plain WordPress installation with version 3.5.2, this results in more than 3.000 call graphs. The call graphs that Doxygen generates only contain function and methods that are in the code base itself and dynamic invocation functions such as `call_user_func` and `call_user_func_array` are ignored. This means that whenever a function or method used dynamic invocation to call another method, these calls are not included. The resulting call graphs are therefore unsound.

To see how our transformation tool alters the accuracy of static analysis, we compare the call graph sizes of an normal WordPress installation and a transformed installation. We counted all the nodes and all the edges in the 3.077 generated call graphs. We did this by analyzing the DOT files created by Doxygen.

Our original WordPress installation has 64.743 nodes and 108.704 edges.

Our transformed WordPress installation has 87.886 nodes and 137.103 edges.

This shows us that the number of nodes and edges for call graphs increases after transforming the code. This of course is not a surprise, since instances of dynamic invocation where previously ignored but the resulting invocations are made explicit in the transformed code. Since the original call graphs did not contain calls made by dynamic invocation and the graphs of the transformed code did, we can say that these graphs give a better representation of the call flow and therefore is more accurate.

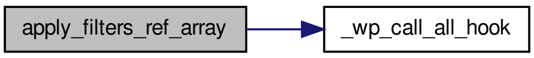

FIGURE 5.3: Call graph from Doxygen for orignal version

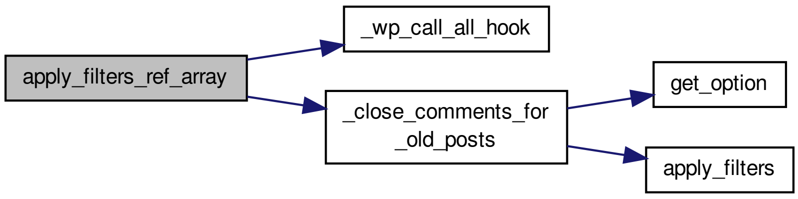

FIGURE 5.4: Call graph from Doxygen for transformed version

5.2.1 Threats to Validity

For some unknown reason DOXYGEN generated around 100 call graphs less for our transformed version of WORDPRESS compared to the original version. To mitigate for this difference, we only compared the call graphs that were generated for both versions. Even though we compensated for this difference, it still makes our results less reliable.

5.2.2 Conclusion

Transforming the source code of WORDPRESS by replacing dynamic invocations with explicit invocations results in more accurate call graphs generated DOXYGEN. The results of the already unsound analysis done by DOXYGEN were enhanced thanks to the additions of call graph nodes and edges of functions and methods that were called via dynamic invocation.

Chapter 6

Evaluation of Performance Improvements

This chapter describes an evaluation of the performance improvements gained from using the implemented tool to replace occurrences of dynamic invocation with explicit function calls.

6.1 Isolated Dynamic Invocation Performance

To get an idea of the performance impact of PHP's `call_user_func` and `call_user_func_array` functions we have profiled isolated calls to these functions.

Our first benchmark script contained the following code:

```
call_user_func("func");
```

Running this piece of code 1,000,000 times showed us that the overhead of `call_user_func` was 2,532,000 microseconds CPU time in total, which is an average of 2.5 microseconds per invocation. This is negligible.

Our second benchmark script contained the following code:

```
call_user_func_array("func", array());
```

Running this piece of code 1,000,000 times showed us that the overhead of `call_user_func` was 2,639,000 microseconds CPU time in total, which is an average of 2.6 microseconds per invocation. This is also negligible.

6.2 Performance in Real World Code

Since previous microbenchmarks show that the performance penalty of dynamic invocation is minimal, further investigation into performance improvements in real code is unnecessary. Because the overhead of dynamic invocation is so small, on average 2.6 microseconds, the impact of such a call in the context of a HTTP request which is measured in tens or hundreds of milliseconds is not noticeable.

6.3 Threats to Validity

The small impact that was measured with our microbenchmark let us to believe that further investigation into possible performance improvements thanks to the reduction of dynamic invocation was unnecessary. However, it could be possible that the transformed source code, due to its more static nature, is more suitable for optimization using static analysis techniques used in interpreters and compilers. We however did not verify this.

6.4 Conclusion

The replacement of dynamic invocation with explicit invocation does not directly result in significant preformance improvements.

Chapter 7

Conclusions & Future Work

This chapter describes the conclusions and possible future work.

7.1 Conclusions

PHP is full-featured scripting language equipped with some useful dynamic features. Dynamic features give developers a lot of flexibility by letting them delegate program behavior to the runtime. One example of a dynamic feature is dynamic invocation, which let you pass in the function or method to be called as a string. However, usage of the dynamic features can have a negative influence on the analyzability and performance of a program. We observed that an important reason for the use of dynamic invocation is to facilitate configuration and plugin functionality. Because configuration and plugin installation tends to stabilize, we state that there is a moment where most dynamic invocation behavior is static. Using dynamic analysis, done with the help of trace analysis, we are able to reduce the number of dynamic features used, especially dynamic invocation. We can do this by capturing execution traces, analyzing these traces and use the gained insights of the usage patterns of occurrences of dynamic invocation to generate static code with the same semantics. To be sure that the static replacements are as complete as possible we leverage the assumption made earlier, that the behavior of dynamic invocation at a certain moment in time is static and could therefore in theory be completely traced. The source code transformation eventually leads to code that can be analyzed more easily using existing types of static analysis. We showed that simple call graph analysis can get more accurate representation of real world call flow behavior. We also looked at the possible performance increase gained by reducing dynamic invocation, however the direct performance impact of dynamic invocation appeared to be so small that it is negligible. The approach of using dynamic analysis to replace dynamic feature use cases with static equivalents for more accurate unsound analysis looks promising, but dynamic analysis is always depended on the level of completeness of the traced execution. You only know what you have traced and you only trace what is executed, but there are situations were all you need to know is what is actually executed.

7.2 Future Work

7.2.1 Partial Evaluation

When transforming the dynamic invocation to explicit method and function calls, we chose to not look at the arguments that are being passed to the functions or methods themselves. However, this information is also captured during the tracing phase and could be used to generate specialized functions for the mostly used arguments. This could lead to performance improvements.

7.2.2 Data-flow Analysis

In addition to the hybrid approach we took were we used execution traces to observe what data is passed into the functions we were interested in, data-flow analysis could bring extra insight into possible values. Combining the information about how variables are being set with the data gathered using traces could lead to more precise approximations of the original program semantics and therefore more accurate static analysis.

7.2.3 Other Dynamic Features

Similar to dynamic invocation were the name of the item which you refer to can be variable are things like variable variables, variable methods and variable functions. You can for example call a function like this: `$funcName()` where `$funcName` is a variable that contains the name of the function which is to me be invoked. This family of variable constructs could be investigated and resolved in the same manner as we did in this research.

Bibliography

[1] PHP: History of PHP. http://www.php.net/manual/en/history.php.php. Accessed: 2013-7-16.

[2] Usage Statistics and Market Share of PHP for Websites, July 2013. http://w3techs.com/technologies/details/pl-php/all/all. Accessed: 2013-07-08.

[3] PHP: call_user_func. http://php.net/manual/en/function.call-user-func.php. Accessed: 2013-07-08.

[4] PHP: eval. http://www.php.net/manual/en/function.eval.php. Accessed: 2013-7-16.

[5] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2574-1. doi: 10.1109/SP.2006.29. URL http://dx.doi.org/10.1109/SP.2006.29.

[6] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP95 Object-Oriented Programming, 9th European Conference, arhus, Denmark, August 711, 1995*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60160-9. doi: 10.1007/3-540-49538-X_5. URL http://dx.doi.org/10.1007/3-540-49538-X_5.

[7] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0. URL http://dl.acm.org/citation.cfm?id=2032497.2032503.

[8] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. Eval begone!: semi-automated removal of eval from javascript programs. *SIGPLAN Not.*, 47(10):

607–620, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384660. URL `http://doi.acm.org/10.1145/2398857.2384660`.

[9] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. *SIGPLAN Not.*, 44(10):283–300, October 2009. ISSN 0362-1340. doi: 10.1145/1639949.1640110. URL `http://doi.acm.org/10.1145/1639949.1640110`.

[10] Mark Hills, Paul Klint, and Jurgen Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. 2013.

[11] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The hiphop compiler for php. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 575–586, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384658. URL `http://doi.acm.org/10.1145/2384616.2384658`.

[12] Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1916–1923, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. doi: 10.1145/1529282.1529709. URL `http://doi.acm.org/10.1145/1529282.1529709`.

[13] WordPress Blog Tool, Publishing Platform, and CMS. `http://wordpress.org/`. Accessed: 2013-8-7.

[14] WordPress now powers 18.9% of the Web, has over 46m downloads, according to founder Matt Mullenweg - The Next Web. `http://thenextweb.com/insider/2013/07/27/wordpress`. Accessed: 2013-8-7.

[15] Xdebug - Debugger and Profiler Tool for PHP. `http://xdebug.org/`. Accessed: 2013-7-10.

[16] CasperJS, a navigation scripting and testing utility for PhantomJS. `http://casperjs.org/`. Accessed: 2013-7-10.

[17] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3793-1. doi: 10.1109/SCAM.2009.28. URL `http://dx.doi.org/10.1109/SCAM.2009.28`.