On Using Static Analysis to Detect Type Errors in PHP Applications

EPFL-REPORT-147867

Etienne Kneuss, Philippe Suter, and Viktor Kuncak

`firstname.lastname@epfl.ch`
EPFL School of Computer and Communication Sciences, Lausanne, Switzerland

Abstract. We describe our experience in using abstract interpretation to analyze applications written in PHP. Our work focuses on reconstructing type information from mostly unannotated code. We present the abstract domain of our analysis, focusing on the features that improve analysis precision. We have implemented our approach as a tool that supports the full specification of PHP 5. We describe several bugs that we were able to find in deployed web applications.

1 Introduction and Background

PHP is a very popular scripting language. PHP scripts are behind many web sites, including wikis, content management systems, and social networking web sites. It is notably used by major web actors, such as Wikipedia[1], Facebook[2] or Yahoo [Rad06]. Unfortunately, it is very easy to write PHP scripts that contain errors. Among the PHP features that are contributing to this fact is the lack of any static system for detecting type or initialization errors. PHP files being usually compiled for every executions, the speed requirement on the compiler makes it impossible to do any smart analysis during that phase.

This paper presents Phantm[3], a static analyzer for PHP 5 based on abstract interpretation [CC77a]. Phantm is an open-source tool written in Scala and available from `http://github.com/colder/phantm`. It contains a robust parser that passes 10'000 tests from the PHP test suite, and a static analysis algorithm for type errors. Phantm uses an abstract interpretation domain that approximates values of variables for both simple and structured types, such as arrays and objects. Phantm is flow-sensitive, supporting a form of typestate [DF01,DF04,FD02,BA05,FL03,FGRY03,FYD$^+$06,LKR05,SY86], which is natural given that the same PHP variable can have different types at different program points.

Phantm supports a large number of PHP constructs in their most common usage scenarios, with the goal of maximizing the usefulness of the tool. It incorporates precision-enhancing support for several PHP idioms that we frequently

[1] `http://www.mediawiki.org/wiki/MediaWiki`
[2] `http://developers.facebook.com/hiphop-php/`
[3] PHp ANalyzer for Type Mismatch

encountered and for which our initial approach was not sufficiently precise. A few other features, such as generic error handlers for undefined methods PHANTM reports as bad practices instead of attempting to abstract the complex behavior of the PHP interpreter.

PHANTM analyzes each function separately, but uses PHP documentation features to allow users to declare types of function arguments. It also comes with detailed type prototype information for a large number of library functions, and can be very helpful in annotating existing code bases. By providing additional flexibility that goes beyond simple type systems, we expect PHANTM to influence future evolution of the language and lead to much more reliable applications.

We have applied PHANTM to three substantial PHP applications. The first application is a webmail client used by several thousand users. The second is the popular DokuWiki software[4], and the third is the feed aggregator library SimplePie.[5] Using PHANTM, we have identified a number of errors in these applications.

In the rest of the paper, we illustrate the capabilities of PHANTM through a number of examples, describe PHANTM's abstract interpretation domain, describe main aspects of the implementation, and discuss our experience in using the tool to find real errors.

2 Example

PHP has a dynamic typing policy: types are not declared statically, and variables can adopt various types at different times, depending on the values assigned to them. The basic types are booleans, integers, floating point numbers, strings, arrays and objects. There is also a null type for undefined values and a special type for external resources such as file handlers or database connections. Variables are not declared. Reading from an uninitialized variable results in null.

PHP arrays are essentially maps from integers and strings to arbitrary values. For instance, the following is a valid definition:

```
$arr = array("one" ⇒ 1, −1 ⇒ "minus one", 3 ⇒ 3.1415);
```

After this assignment, $arr is an array defined for the keys "one", -1 and 3. Contrary to many programming languages, PHP arrays are by default passed by value.

As of PHP 5, the object model is rather standard. Classes declare methods and (possibly static) fields, and can inherit from at most one another class. Fields can be added at runtime by simply writing to undeclared ones, so in this sense objects are maps as well. Objects are always passed by reference, unless explicitly cloned.

A small example. We illustrate some of the challenges in applying type reconstruction to PHP programs and show how PHANTM can tackle them. Consider the following code:

[4] http://www.dokuwiki.org
[5] http://simplepie.org

```
$conf["readmode"] = "r";
$conf["file"] = fopen($inputFile, $conf["readmode"]);
$content = fread($conf["file"]);
fclose($conf["file"]);
```

First, note that several values of different type are stored in an array. To check that the call to the library function **fopen** is correctly typed, we need to be able to establish that the value stored in $conf['readmode'] is a string. This immediately points to the fact that our analyses cannot simply abstract the value of $conf as "any array", as the mapping between the keys and the types of the value needs to be stored. On this code, PHANTM correctly concludes that the entry for the key "readmode" always points to a string.

The function **fopen** tries to open a file in a desired mode and returns a pointer to the file –a resource, in PHP terminology– if it succeeded, and the value **false** otherwise. To properly handle this fact, PHANTM encodes the result of the call as having the type "any resource *union* **false**". Because **fread** expects a resource only, PHANTM will display the following warning message:

```
Potential type mismatch. Expected: Array[file => Resource, ...], found:
Array[file => Resource or False, ...]
```

This warning points to the fact that the case when the file cannot be opened is not handled properly. Although **fclose** expects only a resource as well, our tool will *not* emit a second warning for the fourth line. The reason is that whenever PHANTM detects a type mismatch, it applies *type refinement* on the problematic variable, assuming that the intended type was the one expected rather than the one found. In many cases, this eliminates or greatly reduces the number of warnings for the same variable.

We can change the code to properly handle failures to open the file as follows:

```
$conf["readmode"] = "r";
$conf["file"] = fopen($inputFile, $conf["readmode"]);
if($conf["file"]) {
  $content = fread($conf["file"]);
  fclose($conf["file"]);
}
```

Now that the calls to **fread** and **fclose** are guarded by a check on $conf["file"], PHANTM determines that their argument will never evaluate to **false** and accepts the program as type correct.

3 Abstract Domain

Concrete states. We model runtime values as elements from disjoint sets corresponding to the possible types (see Figure 1). A concrete program state is characterized by a mapping from a set of constant strings to values as well as a heap state. A heap state is simply a set of mappings from object references to object states, where an object state is a mapping from a set of constant strings, namely the names of fields, to values. Object methods are not modelled by the state. We view methods as global functions that take as an extra parameter the receiver ($this). Such approach is possible in part because method definitions are static.

$$\begin{aligned}
\mathsf{V} &= \{\mathsf{True}, \mathsf{False}, \mathsf{Null}\} \cup \mathsf{Ints} \cup \mathsf{Floats} \cup \mathsf{Strings} && \text{values}\\
&\quad \cup \mathsf{Maps} \cup \mathsf{Objs} \cup \mathsf{Resources} &&\\
\mathsf{Maps} &= (\mathsf{Ints} \cup \mathsf{Strings}) \hookrightarrow \mathsf{V} && \text{maps}\\
\mathsf{Tags} &= \{\text{StdClass}, \text{all classes defined in the program}\} &&\\
\mathsf{H} &= \mathsf{Objs} \hookrightarrow (\mathsf{Tags} \times (\mathsf{Strings} \hookrightarrow \mathsf{V})) && \text{heap states}\\
\mathsf{S} &= (\mathsf{Strings} \hookrightarrow \mathsf{V}) \times \mathsf{H} && \text{program states}
\end{aligned}$$

Fig. 1. Characterization of the concrete states. $A \hookrightarrow B$ denotes all partial functions from A to B.

$$\begin{aligned}
\mathsf{V}^\sharp &= \{\mathsf{Undef}^\sharp, \mathsf{True}^\sharp, \mathsf{False}^\sharp, \mathsf{Null}^\sharp, \mathsf{Int}^\sharp, \mathsf{Float}^\sharp, \mathsf{String}^\sharp, && \text{abstract values}\\
&\quad \mathsf{Resource}^\sharp\} \cup \mathsf{Maps}^\sharp \cup \mathsf{Objs}^\sharp &&\\
\mathsf{Maps}^\sharp &= (\mathsf{Ints} \cup \mathsf{Strings} \cup \{?\}) \hookrightarrow \mathsf{V}^\sharp && \text{abstract maps}\\
\mathsf{H}^\sharp &= \mathsf{Objs}^\sharp \hookrightarrow (\mathsf{Tags} \times (\mathsf{Strings} \hookrightarrow \mathsf{V}^\sharp)) && \text{abstract heap states}\\
\mathsf{S}^\sharp &= (\mathsf{Strings} \hookrightarrow \mathsf{V}^\sharp) \times \mathsf{H}^\sharp && \text{abstract program states}
\end{aligned}$$

Fig. 2. Definition of the abstract domain.

$$\begin{aligned}
\beta(\mathsf{Undef}^\sharp) &= \{\mathsf{Null}\},\ \beta(\mathsf{Null}^\sharp) = \{\mathsf{Null}\}\\
\beta(\mathsf{True}^\sharp) &= \{\mathsf{True}\},\ \beta(\mathsf{False}^\sharp) = \{\mathsf{False}\},\ \beta(\mathsf{Int}^\sharp) = \mathsf{Ints}\\
\beta(\mathsf{t} \in \mathsf{Tags}) &= \mathsf{t},\ \beta(\mathsf{i} \in \mathsf{Ints}) = \mathsf{i},\ \beta(\mathsf{s} \in \mathsf{Strings}) = \mathsf{s}\\
\beta(\mathsf{m}^\sharp \in \mathsf{Maps}^\sharp) &= \{\mathsf{m} \mid \forall (\mathsf{k} \mapsto \mathsf{v}) \in \mathsf{m}.\\
&\qquad (\exists (\mathsf{k}^\sharp \mapsto \mathsf{v}^\sharp) \in \mathsf{m}^\sharp\ .\ \mathsf{k} \in \beta(\mathsf{k}^\sharp) \wedge \mathsf{v} \in \beta(\mathsf{v}^\sharp)) \vee\\
&\qquad ((\neg\exists (\mathsf{k}^\sharp \mapsto \mathsf{v}^\sharp) \in \mathsf{m}^\sharp\ .\ \mathsf{k} \in \beta(\mathsf{k}^\sharp)) \wedge (\exists (? \mapsto \mathsf{v}^\sharp) \in \mathsf{m}^\sharp\ .\ \mathsf{v} \in \beta(\mathsf{v}^\sharp)))\}\\
\beta(\mathsf{o}^\sharp \in \mathsf{Objs}^\sharp) &= \{\mathsf{o} \in \mathsf{Objs} \mid \mathsf{o} \text{ was allocated at program point } \mathsf{o}^\sharp\ \}\\
\beta(\mathsf{h}^\sharp \in \mathsf{H}^\sharp) &= \{\mathsf{h} \mid \forall (\mathsf{o} \mapsto (\mathsf{t}, \mathsf{m})) \in \mathsf{h}\ .\ \exists (\mathsf{o}^\sharp \mapsto (\mathsf{t}, \mathsf{m}^\sharp)) \in \mathsf{h}^\sharp .\ \mathsf{o} \in \beta(\mathsf{o}^\sharp) \wedge \mathsf{m} \in \beta(\mathsf{m}^\sharp)\}\\
\gamma(\mathsf{s}^\sharp) &= \beta(\mathsf{m}^\sharp) \times \beta(\mathsf{h}^\sharp) \ \text{ for } \mathsf{s}^\sharp = (\mathsf{m}^\sharp, \mathsf{h}^\sharp) \in \mathsf{S}^\sharp
\end{aligned}$$

Fig. 3. Concretization function. The concretization for $\mathsf{Float}^\sharp$, $\mathsf{String}^\sharp$ and $\mathsf{Resource}^\sharp$ similar to the case for $\mathsf{Int}^\sharp$.

Abstract states and concretization. Our abstract domain is presented in Figure 2. Figure 3 describes the meaning of abstract type elements using a concretization function γ. Most scalar types are abstracted by a single value, with booleans being the exception. String and integer constants are abstracted by their precise value only when they serve as keys in a map. In maps, we use the special value ? to denote the set of keys in a map that are not otherwise represented by a constant. For example, to denote all maps where the key "x" is mapped to an integer and all other keys are undefined we use the abstract value

$$\mathsf{Map}^\sharp[\text{"x"} \mapsto \mathsf{Int}^\sharp, ? \mapsto \mathsf{Undef}^\sharp]$$

We use allocation-site abstraction [CWZ90] for objects. Whereas Objs represents the set of possible memory addresses in the heap, $\mathsf{Objs}^\sharp$ represents the set of program points where objects can be created.

PHP does not distinguish between variables that have never been assigned and variables that have been assigned to the value null. However, using null as a value often has a meaning, while reading from unassigned variables is generally an error. To be able to distinguish between these two scenarios, our analysis uses two different abstract values for these two uses and handles them differently in the transfer function. Our analysis thus incorporates a limited amount of history-sensitive semantics.

Our goal is to approximate the set of types a variable can admit at a given program point. To do so, we consider for our abstract domain not only the values representing a specific type (such as $\mathsf{Int}^\sharp$), but also combinations by union of these. We refer to such combinations of abstract values as *union types*, and we use the symbol τ to denote such a type. Even though we could in principle consider arbitrary union of maps, we chose to simplify them by computing them point-wise, thus, for example,

$$\mathsf{Map}^\sharp[\mathsf{k}_1^\sharp \mapsto \tau_1, \mathsf{k}_2^\sharp \mapsto \tau_2, \ldots, ? \mapsto \tau_D] \cup \mathsf{Map}^\sharp[\mathsf{k}_1^\sharp \mapsto \tau_1', \mathsf{k}_3^\sharp \mapsto \tau_3', \ldots, ? \mapsto \tau_D']$$

is approximated with

$$\mathsf{Map}^\sharp[\mathsf{k}_1^\sharp \mapsto \tau_1 \cup \tau_1', \mathsf{k}_2^\sharp \mapsto \tau_2 \cup \tau_D', \mathsf{k}_3^\sharp \mapsto \tau_D \cup \tau_3', \ldots, ? \mapsto \tau_D \cup \tau_D']$$

What is lost is the simplification is the relation between the types for various indices. This is consistent however with the treatment of types for variables: keeping the complete type would amount to a form of dependent analysis for elements of arrays, which we do not do for variables. In other words, just like we analyze variables independently from each other, so do we analyze array entries.

The concretization of a union type $\tau_1 \cup \tau_2$ is naturally given by $\beta(\tau_1) \cup \beta(\tau_2)$. It is straightforward to verify that the set of union types forms a lattice where the partial order corresponds to the notion of subtyping.

In general, a type can be the union of any two types. We denote type unions by the symbol $\cup$. The subtyping relation is naturally extended with the following two rules:

$$\begin{aligned} \tau \sqsubseteq (\tau_1 \cup \tau_2) &\iff \tau \sqsubseteq \tau_1 \vee \tau \sqsubseteq \tau_2 \\ (\tau_1 \cup \tau_2) \sqsubseteq \tau &\iff \tau_1 \sqsubseteq \tau \wedge \tau_2 \sqsubseteq \tau \end{aligned}$$

The subtype relation is defined point-wise for array types:

$$\mathsf{Map}^\sharp[k_1 \mapsto \tau_1, k_2 \mapsto \tau_2, \ldots, ? \mapsto \tau_D] \sqsubseteq \mathsf{Map}^\sharp[k_1 \mapsto \tau_1', k_3 \mapsto \tau_3', \ldots, ? \mapsto \tau_D'] \\ \iff \tau_1 \sqsubseteq \tau_1' \wedge \tau_2 \sqsubseteq \tau_D' \wedge \tau_D \sqsubseteq \tau_3' \wedge \ldots \wedge \tau_D \sqsubseteq \tau_D'$$

Therefore, $\mathsf{Map}^\sharp[? \mapsto \bot]$ and $\mathsf{Map}^\sharp[? \mapsto \top]$ are a subtype and a supertype of all array types, respectively.

Enforcing termination. As in constant propagation, one reason for termination follows from the fact that the set of constants and keys is limited to those syntactically occurring in the program. However, an additional potential for an infinite-height lattice are nested arrays. We thus enforce termination by limiting the array nesting depth to a constant (currently 5). We have found this approach to work well in practice.

Optimistic assumptions on heap manipulations. We assume that each function and method affect distinct parts of the heap. We ignore side effects on objects passed to functions, and assume that functions returning objects always return a fresh instance. This allows us to perform intraprocedural analysis. Note that we do take into account type annotations on parameters from PHP declarations as well as a common documentation format, which allows the developer to improve the precision by additional type annotations. While these assumptions are clearly optimistic, we should note that, given our coarse model for runtime values (abstracting the type information), reasoning only becomes unsound when functions affect the *type* of an object's field. For example, we do not expect a setter method to change the type of a field.

3.1 Transfer Function

For space reasons we only give a brief outline of the abstract transfer function. A compact description of the transfer functions of our analysis in Scala is given in around 1000 lines of Scala source code. [6]

Type refinement. Since the PHP language allows little to no type annotations, it is often the case that types of values are completely unknown before being actually used. In order to reduce the number of false positives generated by consecutive uses of such values, it is crucial that their types get refined along the way. For instance, the following piece of code will only generate one notice:

```
$b = $a + 1;
$c = $a + 2;
```

After the first statement, it is assumed that `$a` is a valid operand for mathematical operations, by refining it with the type $\mathsf{Int}^\sharp \cup \mathsf{Float}^\sharp$. For this purpose, we compute the lattice meet between the type lattice elements corresponding to

[6] File `src/phpanalysis/controlflow/TypeFlow.scala` in the repository at `http://github.com/colder/phantm/`

the current and the expected variable types. To get the idea of this operation, we show a typical computation of the intersection of array types:

$$\mathsf{Map}^\sharp[\mathsf{k}_1^\sharp \mapsto \tau_1, \mathsf{k}_2^\sharp \mapsto \tau_2, \ldots, ? \mapsto \tau_D] \sqcap \mathsf{Map}^\sharp[\mathsf{k}_1^\sharp \mapsto \tau_1', \mathsf{k}_3^\sharp \mapsto \tau_3', \ldots, ? \mapsto \tau_D'] =$$
$$\mathsf{Map}^\sharp[\mathsf{k}_1^\sharp \mapsto \tau_1 \sqcap \tau_1', \mathsf{k}_2^\sharp \mapsto \tau_2 \sqcap \tau_D', \mathsf{k}_3^\sharp \mapsto \tau_D \sqcap \tau_3', \ldots, ? \mapsto \tau_D \sqcap \tau_D']$$

Such type refinement corresponds to deriving an 'assume' statement that is a consequence of successful execution of an operation, and therefore preserves the monotonicity of transfer functions.

Conditional filtering. Type refinement is also used in assume statements implied by various control structures. It is important to note that PHP allows values of every types to be used as boolean conditions, and gives different boolean values to inhabitants of those types. This allows us to do some refinement on the types of values used as boolean conditions. For instance the type **null** can only evaluate to false while integers may evaluate to both true or false (always true except for 0). This is especially useful for booleans, for which we also define **true** and **false** as types. We can then precisely annotate a function returning false on error, and a different type on success, and rely on the type refinement to filter it out only when the error is correctly checked for. In case the type of the value drops to $\perp$ during the refinement, we assume that the branch cannot be taken, allowing us to detect unreachable code.

4 Reporting Type Errors using Reconstructed Types

When the analysis reaches its fixpoint, it has effectively reconstructed possible types for variables at all program points. At this point in time, PHANTM makes a final pass over the program control-flow graph and reports type mismatch. Because transfer functions already perform type refinement, they contain all the necessary information to report type mismatch, and we reuse them to report type errors. PHANTM reports a type mismatch whenever the computed type at a given program point does not match the expected type. PHANTM has a number of options to control the verbosity of its warnings and errors.

5 Implementation Highlights

5.1 Overview

This tool can be separated into four parts: lexing, parsing, structural analysis and data flow analysis. Lexing has been implemented using JFlex[7]. Since PHP uses flex, the lexer description could be translated directly from the original one with only few modifications. The parser is implemented using a modified version of CUP[8]. Again, it was possible to import the original yacc version without many modifications. The last two parts are implemented in Scala [OSV08] and represent the core of PHANTM.

[7] http://jflex.de/
[8] http://www2.cs.tum.edu/projects/cup/

5.2 Features

Built-in Support for Important APIs. By default, PHP comes with a very dense library of functions and classes. In fact, the main extensions that are shipped with PHP consist of more than 2'500 functions and classes. Being able to correctly represent this internal API is a key factor to obtain useful analysis results. This API is stored in an external XML file, allowing easy modifications and also do not require a re-compilation. Additionally, a `--importAPI` command line option is available to specify a list of API files that can be imported into the symbol tables. Along with an external API, PHANTM will extract in-code annotations written in the PHP Documentor[9] format (similar to JavaDoc, for instance):

```
/**
 * @param $a Int
 * @return Array | false
 */
 function foo($a) {
    if ($a < 0)
       return false;
    else
       return range(0, $a); }
 }
```

Based on the various call-sites of the user-defined functions, PHANTM will also be able to generate a corresponding API in XML format. This file can then easily be refined by hand, and imported for subsequent analyses in order to get more precise results.

Include resolutions. In order to allow the analysis of large applications, it is required to automatically resolve **include** and **require** calls. They are mainly used to import definitions or blocks of code in a given scope. A file name is passed as argument as a string. PHANTM will resolve the name for any combination by concatenation of the following:

– string literals
– context-dependant –or magic constants defined by PHP, such as **__FILE__**.
– well-defined constants (similar to C macros)
– some pure functions commonly used as part of include statements such as dirname()

Any other file inclusion will prompt a warning and be ignored. Since constants are defined using the **define** function, some trivial constant propagation techniques are used to determine the value of most constants before data-flow analysis. This means that the CFG will not have to change during the fix-point analysis.

IDE integration. PHANTM exposes many run-time configuration options that allows a focused analysis (e.g. per-file or per-function). Additionally, it can

[9] http://www.phpdoc.org

prompt errors in several formats, making it easy to integrate with various IDEs. For instance, it integrates smoothly with VIM[10], a popular text editor. Being able to look at errors from the context of an editor is a definite advantage when it comes to filtering out false positives, as the context of the error is crucial. By default, Phantm will simply prompt errors and display the corresponding line, highlighting the erroneous part. Since type errors using structural types rapidly become hard to read, it will only include the relevant part of the types that mismatch.

5.3 Design Decisions

Conditional declarations. In PHP, functions or classes can be declared conditionally. This naturally creates problems for our analyses, but also induces performance hits on servers equipped with so-called opcode cachers. Those cachers are responsible for storing the intermediate –or compiled– version of each file, function and class, the goal being to speed up the process by reducing the number of compilations required per request. This cannot be done easily if those declarations are conditional. Phantm will consider the first declaration to be global and discard future declarations. We argue that this behavior will be valid for analyzing most code-bases. Indeed, the primary reason for using conditional declarations is to provide a userland implementation of an internal class/function, if the application is being run by a version of PHP in which it is not defined.

References. References will be ignored as they introduce too many aliasing problems. The usage of references is in most cases discouraged. As effect, reference usage is more and more sparse, up to a point where providing reference support to correctly analyze such a small fraction of the code-bases would not justify the increased complexity.

Dynamic object properties. PHP allows dynamic references to an object property using a variable or expression (e.g. $name = "a"; $obj->$name instead of $obj->a). This is usually considered bad practice and will result in a global access/modification of the object

Dynamic variables. PHP allows to reference a variable using either a variable, or an expression[11]: ($$var or ${'prefix'.$name.foo()}). This is considered bad practice and will be ignored, as treating it as a global access/modification would introduce too many false positives.

Assignments in conditional expressions. Assignments in PHP return the value assigned, they are hence valid expressions inside conditional expressions. However, history tells us that most of the time, this is an actual typographic error replacing the comparison operator **==** with the assignation operator **=**. This tool will thus emit a warning if such expression is found inside an **if()** or

[10] `http://www.vim.org/`

[11] PHP Variable variables: `http://php.net/variables.variable`

for() condition. We exclude **while()** on purpose as there is a common use-case where assignations are done directly inside the **while()** expression.

Dynamic code execution. Phantm will ignore all kind of dynamic code execution, such as `eval()` or `create_function()` which accepts actual code as string arguments. Using these methods are generally discouraged.

Autoload. PHP provide a way to load classes on-demand. This is done by defining a set of functions that will be called whenever an undefined class is being referenced. This feature would be close to impossible to model and hence will be ignored by Phantm. This limitation is however trivial to circumvent. Indeed, one can simply prepend a file including all undefined classes statically.

6 Evaluation

We evaluated Phantm on three benchmarks. The first one is an email client which we will call WebMail, similar in functionality to IMP.[12] It has been in production for several years. There are currently over 5000 users registered to the service. WebMail was written in PHP 4.1 and has not evolved much since its launch. The source code is not public but has kindly been made available to us by the development team. Our second benchmark is the popular open source wiki project DokuWiki and the third benchmark is SimplePie, an open source library to manage the aggregation of RSS and Atom news feeds.

6.1 Analysis of WebMail

Issues identified. We discuss some of the bugs and other issues we were able to find in the source code of WebMail.

– By running Phantm on all PHP files in the document root, that is all files accessible from the web, we identified a vulnerability in the file `lib.php`. It contains the following line of code:

```
require_once("$inc_dir/functions.php");
```

Phantm reports a warning that the variable `$inc_dir` is not defined. The problem is that `lib.php` is not supposed to be accessed directly. (No link in WebMail points to it.) The file is only meant to be included from other, accessible, files. In particular, it is normally included in a context where the variable `$inc_dir` is defined. Because in PHP 4.1 variables passed to scripts through HTTP queries are directly available in the code, this code is subject to potential attacks. By using a crafted HTTP query on `lib.php` such as

```
GET /lib.php?inc_dir=http://example.com/attacker_script.phps?
```

an attacker can have an arbitrary PHP script downloaded and run on the server where WebMail is deployed. This can lead to the exposure or corruption of important data, such as root passwords or email account credentials.

[12] http://www.horde.org/imp/

– In a function handling the conversion from one string format to another, Phantm emitted a warning on the following line:

```
$newchar = substr($newcharlist, strpos($charlist, $char), 1);
```

The warning indicated that **substr**() expects a string as its second argument, but that in this case the type False ∪ String had been found. The library function **strpos**() searches for the index of first occurrence of a string within another one and returns **false** if it could not find it. The developers were assuming that $charlist would always contain $char. However, a closer inspection of the code revealed that this was not always the case. We found that because of this bug, some passwords were improperly stored, potentially resulting in some email accounts to be inaccessible from WebMail. While in this case Phantm did not point us directly to the source of the bug (the value of $charlist), we could most certainly not have found it without the tool. We note that this bug had not been uncovered so far, despite the fact that WebMail has over 5000 registered users.[13]

– In several places, two distinct functions were called with too many arguments. This was apparently the result of an incomplete refactoring during the development. Although these extra arguments did not cause any bug (they are silently ignored by the PHP interpreter), they were clearly errors. Subsequent changes to the code could also have led to new bugs because of them.

– In a file containing definitions for the available languages, Phantm reported a warning on the second of the following lines:

```
$dict["en"]["fr"]="anglais";
$dist["en"]["de"]="englisch";
```

The first line is well formed and stores the translation for “English” in French. The second line stores the translation in German but the name of the array is misspelled. The assignment is valid in PHP and creates an array where the entry for "en" is defined to be the array { "de" → "englisch" }. Because the entry $dist["en"] is not defined to be an array previously in the code, though, Phantm flags this as bad style and reports a warning. This bug was never reported as its only consequence was that German-speaking users saw the default spelling (“English”) instead of the correct translation in some menus.

– There were several warnings for code such as $i = $str * 1, which essentially casts a string into an integer by using the implicit conversion triggered by the multiplication. Although it is not incorrect, it is flagged as bad style.

Sources of false positives. What we found to be the main source of false positives in the analysis of WebMail are arrays built from SQL queries. For instance, the following code

[13] A partial explanation is that a bug report could only lead to the discovery of the problem if the user included his password, and this is discouraged by the fact that the bug tracking system is public.

```
$row = mysql_fetch_row(mysql_query("SELECT name FROM 'users' where id='12'"));
```

will assign to the variable $row an array where the key "name" is well-defined and points to a value of the right type. Our tool currently cannot infer such information and a subsequent access to $row["name"] will therefore trigger a warning.

6.2 Analysis of DokuWiki

Issues identified.

- We found multiple instances where the code relied on implicit conversions. Even though this is a commonly used feature of PHP, relying on them often highlights programming errors. For example, the following line

  ```
  $hid = $this→_headerToLink($text,'true');
  ```

  calls the method _headerToLink which is defined to take a boolean as its second argument, not a string. This code is not wrong per se, as the string "true" will evaluate to true which is likely to be the intended behavior, but "false" would evaluate to true as well.
- Keeping code documentation synchronized with the code itself is often problematic. As an illustration of this fact, PHANTM uncovered multiple errors in the annotations. An example of an incorrect annotation was:

  ```
  /* @return String The encrypted output */
  function _encryptBlock($L, $R) {
    // ...
    return array('L' ⇒ $R, 'R' ⇒ $L);
  }
  ```

  Since DokuWiki uses the PHPDocumentor format, PHANTM takes the annotations into account and checks that they are correct. The last line being the only **return** statement, it is clear that the function always returns an array rather than a string, as declared.
- We found a potential bug resulting from an unchecked file operation in the following function:

  ```
  function bzfile($file) {
      $bz = bzopen($file,"r");
      while (!feof($bz)){
          $str = $str . bzread($bz,8192);
      }
      bzclose($bz);
      return $str;
  }
  ```

  If **bzopen** fails to open the file denoted by $file, it will return **false** and as a consequence the call to **feof** will always return **false**, thus resulting in an infinite loop.

Sources of false positives. In the case of DokuWiki, the principal source of false positives came from the representation of the type assigned to the iterator in **foreach** loops. Most of the time, the program logic seems to be ensuring that the keys can only range over integers or only over strings, but because of previous dynamic updates on the iterated arrays, PHANTM cannot exclude string or integers and assumes both can be present.

6.3 Analysis of SimplePie

Issues identified.

– The following line of code assumes different operator precedence rules than those used by PHP:

```
if (... && !($file→method & SP_FILE_SRC_REMOTE === 0  ...))
```

The code first compares the constant SP_FILE_SRC_REMOTE to 0 –this is always **false**– and then computes the bitwise conjunction, while the goal is clearly to check whether a flag is set in $file→method. This is obviously wrong, as the intended behavior is to check whether a particular flag is not set on that variable. PHANTM found that mistake by reporting that the right-hand side of **&** is a boolean value, and that an integer was expected (because bitwise operations are only valid on integers). The same pattern is used in several places and has been successfully reported each time.

– The following code was also flagged by PHANTM as incorrect:

```
if (... && strtolower(trim($attribs['']['mode']) == 'base64'))
```

The warning indicated that **strtolower**() expects a string as an argument, not a boolean. A close inspection of the statement shows that the right parenthesis of the call to **strtolower** is misplaced, in effect computing the lower case version of a boolean.

Sources of false positives. Our main concern with SimplePie is the omnipresent use of high-dimensionality arrays to organize data. Code such as the following is ubiquitous:

```
if(isset($this→data['child'][SP_NS_RDF]['RDF'][0]['child'][SP_NS_RSS_10]['textinput'])) {
  ...
}
```

Even with refinement, the use of such constructs with different indices at various places is likely to generate a lot of false positives. It is also worth noting that those false positives would also occur on fully annotated code, since PHPDocumentor does not support the specification of structural array types. As a consequence, $this→data can only be annotated as being an array.

6.4 Summary

We summarize the results of our evaluation in Figure 4. In general, we were able to quickly identify the source of problems using hints from Phantm, despite the fact that we were initially not familiar with any of the applications. We also found that the running time of the analysis was never problematic, even on a relatively large project such as DokuWiki.

	Lines of code	Warnings	Bugs	Bad style	Analysis Time
DokuWiki	31486	368	3	48	23 s.
WebMail	3850	126	5	32	21 s.
SimplePie	15003	294	4	76	15 s.
Total	*50339*	*788*	*12*	*156*	*59* s.

Fig. 4. Summary of evaluation results. "Bugs" is the number of problems identified that can lead to crashes or other defects, or that are blatant mistakes. "Bad style" indicates instances where the code relies on implicit type castings that could be simply rewritten into explicit ones.

7 Related Work

Abstract interpretation for type inference. Our work performs type inference using a abstract interpretation, resulting in a flow-sensitive static analysis. Early work on the use of abstract interpretation for type analysis is [CC77b]. A systematic analysis of type analyses of different precision is presented in [Cou97].

Static analysis of PHP. Existing work on static analysis of PHP primarily focused on specific security vulnerabilities. Pixy [JKK06a, JKK06b] is a static analysis tool checking for security vulnerabilities such as cross site scripting (XSS) or SQL injections, which remain the main vectors for attacks on PHP applications. However, it does not support PHP5. Wassermann and Su [WS07] present work on statically detecting SQL injections.

It is only recently that some work have been focusing on static analysis of types in PHP applications. Notably, the Facebook HipHop project[14] is relying on a certain amount of type analysis in order to optimize the PHP runtime. In essence, HipHop tries to find the most specific type used in order to map it to a native C++ type. In case such a type cannot be inferred, it simply falls back to a generic type. The recently released tool PHPLint[15] has as the goal detecting bugs through type errors. Even if its goal is close to the present work, our tool has a much more precise abstract domain, and therefore reports many fewer spurious warnings.

[14] http://github.com/facebook/hiphop-php/

[15] http://www.icosaedro.it/phplint/

Type inference for other languages. Researchers have also considered flow-sensitive type inference in other languages. Soft typing approach has been explored primarily in functional languages [Fag92,AWL94], and supports even first class functions, but is not flow-sensitive and does not support value array types. In [KRCF05] researchers present an analysis of Cobol programs that recovers information corresponding to tagged unions. The work on the C programming language [JMX07, CR99] deals with a language that allows subtle pointer and address arithmetic manipulations, but already contains significant static type information. PHP is a dynamically type safe language in that the run-time system stores dynamic type information, which makes e.g. ad-hoc tagged unions often unnecessary. On the other hand, PHP language by itself provides no static type checking, which makes the starting point for analysis lower. In addition to considering a different language, one of the main novelties of our work is the support for not only flat types but also heterogeneous maps and arrays. Our Phantm tool is publicly available, and we report verifiable experimental results on significant code bases, including popular software whose source code is publicly available.

8 Conclusion

We have presented a framework for static analysis of PHP code based on abstract interpretation. Our analyses attempt to reconstruct type information from untyped code in order to statically detect potential errors. Using this approach, we recover some of the benefits of statically typed programming languages. We have implemented these ideas as a tool and have evaluated it over three existing PHP projects, totalling over 50'000 lines of code. We were able to detect several errors, ranging from simple typos with little or no consequence to security vulnerabilities and functional defects. Although the code contained few or no type annotations, we were able to obtain meaningful diagnosis messages that quickly led us to the critical program points.

References

AWL94. Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM POPL*, pages 163–173, New York, NY, 1994.

BA05. Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In Harald C. Gall, editor, *Proceedings of ESEC-FSE '05*, pages 217–226, September 2005.

CC77a. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

CC77b. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

Cou97. Patrick Cousot. Types as abstract interpretations. In *Proc. 24th ACM POPL*, 1997.

CR99. Satish Chandra and Thomas Reps. Physical type checking for C. In *Workshop on Program analysis for software tools and engineering (PASTE)*, 1999.

CWZ90. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.

DF01. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

DF04. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proc. 18th ECOOP*, June 2004.

Fag92. Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, 1992.

FD02. Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM PLDI*, 2002.

FGRY03. John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. In *Int. Symp. Static Analysis*, volume 2694 of *LNCS*. Springer, 2003.

FL03. Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

FYD$^+$06. Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *ISSTA'06*, 2006.

JKK06a. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

JKK06b. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Programming Languages and Analysis for Security (PLAS)*, 2006.

JMX07. Ranjit Jhala, Rupak Majumdar, and Ru-Gang Xu. State of the union: Type inference via Craig interpolation. In *TACAS*, pages 553–567, 2007.

KRCF05. Raghavan Komondoor, Ganesan Ramalingam, Satish Chandra, and John Field. Dependent types for program understanding. In *TACAS*, 2005.

LKR05. Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *6th Int. Conf. Verification, Model Checking and Abstract Interpretation*, 2005.

OSV08. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

Rad06. Michael J. Radwin. PHP at Yahoo! http://public.yahoo.com/~radwin/talks/php-at-yahoo-mysqluc2006.ppt, 2006.

SY86. Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.

WS07. Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, 2007.