Runtime Instrumentation for Precise Flow-Sensitive Type Analysis

Etienne Kneuss, Philippe Suter, and Viktor Kuncak [⋆]

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

Abstract. We describe a combination of runtime information and static analysis for checking properties of complex and configurable systems. The basic idea of our approach is to 1) let the program execute and thereby read the important dynamic configuration data, then 2) invoke static analysis from this runtime state to detect possible errors that can happen in the continued execution. This approach improves analysis precision, particularly with respect to types of global variables and nested data structures. It also enables the resolution of modules that are loaded based on dynamically computed information.
We describe an implementation of this approach in a tool that statically computes possible types of variables in PHP applications, including detailed types of nested maps (arrays). PHP is a dynamically typed language; PHP programs extensively use nested value maps, as well as 'include' directives whose arguments are dynamically computed file names. We have applied our analysis tool to over 50'000 lines of PHP code, including the popular DokuWiki software, which has a plug-in architecture. The analysis identified 200 problems in the code and in the type hints of the original source code base. Some of these problems can cause exploits, infinite loops, and crashes. Our experiments show that dynamic information simplifies the development of the analysis and decreases the number of false alarms compared to a purely static analysis approach.

1 Introduction

It is challenging to apply precise static analysis to realistic software applications; such applications often give results that are less precise than desired. The imprecision stems both from 1) the approximation that is necessary to ensure acceptable analysis performance, and 2) the absence of detailed information about the environment in which the application runs (such as the file system and user inputs). A common pattern that makes static analysis difficult is reading in configuration data from the external environment, then substantially changing the program behavior based on this data: turning certain features on or off, and loading external modules determined by the configuration. A static analysis typically gives very imprecise results in such cases; it can even fail to determine which files

[⋆] This research was supported in part by the Swiss National Science Foundation Grant #120433 "Precise and Scalable Analyses for Reliable Software".

to include in the application code base, making a conservative analysis entirely useless. Whereas a purely dynamic analysis for such software systems is useful, it may entirely miss opportunities for identifying errors by code inspection.

A hybrid approach. To address these difficulties we propose the following hybrid approach: 1) run the application in its environment as usual, in a deployment-like scenario, up to a user-specified point where most configuration data is expected to be known; 2) record the program state at this point; and 3) use the recorded state as the starting point for a static analysis. The values of many configuration variables thus effectively become constant. This improves the analysis, both of data structures and of control-flow, in some cases making the subsequent results vastly more precise.

We believe that such an hybrid analysis approach deserves more attention than it has received so far. Previous approaches in this spirit include symbolic execution from concrete state [12] and explicit-state model checking from concrete state [15]. In this paper, we show the benefits of this hybrid approach for data-flow analysis. We examine the problem of checking for *type errors* in applications written in PHP, a popular dynamically-typed scripting language.

PHP as the language of the web. 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,Facebook[1] or Yahoo.[2] 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.

Our analyzer. This paper presents Phantm[3], a hybrid static and dynamic analyzer for PHP 5. Phantm is an open-source tool written in Scala and available from http://lara.epfl.ch/dokuwiki/phantm. It contains a full parser that passes 10'000 tests from the PHP test suite, a static analysis algorithm for type errors, and a library to save and restore representations of concrete program states. Phantm uses an abstract interpretation domain that approximates both simple and structured values (such as arrays and objects). Phantm is flow-sensitive, handling initialization and supporting a form of typestate [13]. The motivation for this feature is that the same PHP variable can have different types at different program points. Moreover, the analyzer's notion of type also represents certain concrete values manipulated by the program. Flow sensitive analysis of structured values enables Phantm to handle, e.g., frequently occurring code that uses untyped arrays with string keys as a substitute for records.

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 encountered and for which our initial approach was not sufficiently precise. Phantm reports some other features of PHP, such as generic error handlers for undefined methods, as bad practice instead of attempting to abstract

[1] http://wiki.github.com/facebook/hiphop-php/
[2] http://public.yahoo.com/~radwin/talks/php-at-yahoo-mysqluc2006.ppt
[3] PHp ANalyzer for Type Mismatch

the complex behavior of the PHP interpreter. Based on our experience as PHP programmers, we believe that this is a reasonable design decision.

Phantm analyzes each function separately by default 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 standard library functions, and can be very helpful in annotating existing code bases. By providing additional flexibility in annotation and analysis, going beyond simple type systems, we expect Phantm to influence future evolutions of the language and lead to more reliable applications. Phantm also supports context-sensitive analysis of non-recursive functions without annotations.

Leveraging runtime information. Phantm ships with a library that can be used to instrument the analyzed code and thereby improve the precision of error detection. Programs can be annotated to indicate that the static analysis should start at a given execution point, or to collect a trace of dynamically included files. The collected information is then read by the static analysis component which can use it to, for instance, conclude that certain parts of the program are never executed under a given initial configuration, to detect which function declarations are active, and to refine possible types and values of variables.

Experience. 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 SimplePie.[5] Using Phantm, we have identified a number of errors in these applications.

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`.

The *arrays* in PHP are essentially maps from integers and strings to arbitrary values; we thus use the terms *array* and *map* interchangeably. 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 passed by value and not aliased on assignments.

Handling typestate and nested data types. We illustrate some of the challenges in type analysis of PHP programs and show how Phantm tackles them. Consider the following code:

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

```
$inputFile = "template.txt";
$conf["readmode"] = "r";
$conf["file"] = fopen($inputFile, $conf["readmode"]);
$content = fread($conf["file"]);
echo $content;
fclose($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 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, or the value **false** otherwise. To properly handle this fact, PHANTM encodes the result of the call as having the type "any resource *or* **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 indicates that the developer did not handle the case when the file could not be opened. Note that **fclose** also expects only a resource, but PHANTM does 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:

```
$inputFile = "template.txt";
$conf["readmode"] = "r";
$conf["file"] = fopen($inputFile, $conf["readmode"]);
if($conf["file"]) {
  $content = fread($conf["file"]);
  echo $content;
  fclose($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 therefore accepts the program as type correct.

As a special case, PHANTM also detects uninitialized variables and array entries.[6] If we omit the first line in the source above, PHANTM will warn that the first argument of **fopen** is uninitialized, which could be used by an attacker to reveal the content of arbitrary accessible file on the server.

[6] These errors were a major source of vulnerabilities in past PHP versions, because the `register_globals` server configuration option was activated by default.

Using runtime instrumentation. PHP allows the inclusion of dependencies using dynamic paths. The following example, inspired by DokuWiki code, illustrates how such dynamic features rapidly result in a lot of false alarms when analyzed purely statically:

```
$conf = array('version' ⇒ '1.2.3',
              'path_images' ⇒ 'images/',
              'canWrite' ⇒ is_writeable("data/"),
              'path_modules' ⇒ is_dir('local/') ? 'local/' : 'default/');
include 'config.php';
if (empty($modules)) { // default modules
    $modules = array('smiley' ⇒ array('inc/smiley.inc', 'inc/smiley2.inc'),
                     'acronyms' ⇒ array('inc/acronyms.inc'), ); }
foreach($modules as $files) {
   foreach($files as $file) {
      include getFullPath($conf, $file); } }
phantm_collect_state(get_defined_vars()); // record runtime state
function log_msg($msg) {
    global $conf;
    if ($conf['canWrite']) {
        file_put_contents("data/log", $msg, FILE_APPEND); } }
function displaySmiley() {
    global $conf;
    echo $conf['path_images'].$conf['smiley']['image'][':)']; }
```

In this example, the list of modules is configuration-dependent. Also, based on that list of modules, the code includes their associated files using a non-trivial indirection to resolve the path via getFullPath(). Later on, the displaySmiley() function accesses global and module-dependent configuration settings, assuming that they are defined. Such code would be extremely difficult to analyze purely statically without emitting any false positive. In order to analyze the rest of the application, it is crucial to know the exact state of the program after all the modules are initialized. Runtime instrumentation is a natural and convenient way to obtain this information.

Benefits of hybrid analysis in program understanding. Note that with runtime instrumentation, PHANTM will inform the user that the call to file_put_contents() is unreachable, if run in an environment where the `data/` directory is not writeable. As another example, consider the following code:

```
if (is_debug()) { $debug = true; } else { $debug = false; }
phantm_collect_state(get_defined_vars());
...
if ($debug) { ... }
```

PHANTM detects that the final **if** branch is unreachable when the code runs in a non-debug environment. Such warnings help the user understand which regions of code are relevant in a given environment.

3 Data-Flow Analysis for Flow-Sensitive Type Inference

We first outline the data-flow analysis that we use to infer types. Our description applies regardless of whether the analysis starts from the initial program state or from the dynamically recorded state captured as described in Section 4.

Concrete states. As a first step in providing the meaning of our analysis representation, we present our model of runtime values, which are elements of disjoint sets corresponding to the possible types (see Figure 1). A concrete program state contains 1) a map from a set of constant strings (variable names) to values, and 2) a heap. A heap maps object references to object states, where an object state is a map from a set of constant strings (field names) to values.

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

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

$$\begin{array}{rll}
\mathsf{DV}^\sharp = & \{\mathsf{True}^\sharp, \mathsf{False}^\sharp, \mathsf{Null}^\sharp, \mathsf{Int}^\sharp, \mathsf{Float}^\sharp, \mathsf{String}^\sharp, & \text{defined values} \\
& \mathsf{Resource}^\sharp\} \cup \mathsf{Maps}^\sharp \cup \mathsf{Objs}^\sharp \cup \mathsf{Ints} \cup \mathsf{Floats} \cup \mathsf{Strings} & \\
\mathsf{AV}^\sharp = & \{\mathsf{Undef}^\sharp\} \cup \mathsf{DV}^\sharp & \text{all values} \\
\mathsf{V}^\sharp = & \mathcal{P}_{\mathsf{fin}}(\mathsf{AV}^\sharp) \cup \{\top\} & \text{finite unions and top} \\
\mathsf{Maps}^\sharp = & (\mathsf{Ints} \cup \mathsf{Strings} \cup \{?\}) \hookrightarrow \mathsf{V}^\sharp & \text{abstract maps} \\
\mathsf{H}^\sharp = & \mathsf{Objs}^\sharp \hookrightarrow (\mathsf{Tags} \times (\mathsf{Strings} \hookrightarrow \mathsf{V}^\sharp)) & \text{abstract heap states} \\
\mathsf{S}^\sharp = & (\mathsf{Strings} \hookrightarrow \mathsf{V}^\sharp) \times \mathsf{H}^\sharp & \text{abstract program states}
\end{array}$$

Fig. 2. Definition of the abstract domain.

$$\begin{array}{rl}
\beta(\mathsf{Null}) = & \mathsf{Null}^\sharp,\ \beta(\mathsf{False}) = \mathsf{False}^\sharp,\ \beta(\mathsf{True}) = \mathsf{True}^\sharp \\
\beta(\mathsf{i} \in \mathsf{Ints}) = & \mathsf{i},\ \beta(\mathsf{s} \in \mathsf{Strings}) = \mathsf{s},\ \beta(\mathsf{f} \in \mathsf{Floats}) = \mathsf{f} \\
\beta(\mathsf{o} \in \mathsf{Objs}) = & \mathsf{o}^\sharp \in \mathsf{Objs}^\sharp \text{ where } \mathsf{o} \text{ was allocated at site } \mathsf{o}^\sharp \\
\beta(\mathsf{m} \in \mathsf{Maps}) = & \beta_2(\mathsf{m}^\sharp, 0) \\
\beta_2(\mathsf{m} \notin \mathsf{Maps}, i) = & \beta(\mathsf{m}) \\
\beta_2(\mathsf{m} \in \mathsf{Maps}, i < 5) = & \{(\beta(\mathsf{k}) \mapsto \beta_2(\mathsf{v}, i+1)) \mid (\mathsf{k} \mapsto \mathsf{v}) \in \mathsf{m}\} \\
\beta_2(\mathsf{m} \in \mathsf{Maps}, i \geq 5) = & (? \mapsto \mathsf{AV}^\sharp)
\end{array}$$

Fig. 3. Abstraction β of variable values used to define the abstraction function α

Analysis representation and abstraction function. Our abstract domain is presented in Figure 2. We use $\perp$ to denote an empty set of elements of $\mathsf{V}^\sharp$.

Figure 3 describes the meaning of abstract type elements using function β that abstracts the values of values of individual variables. The analysis abstracts boolean, string and integer constants by their precise value when it is known, for instance when they serve as keys in a map. We refer to such precise values as *singleton scalar types*. In maps, we use the special value ? to denote the set of keys 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 $\mathsf{Map}^\sharp[\text{"}\mathsf{x}\text{"} \mapsto \mathsf{Int}^\sharp, ? \mapsto \mathsf{Undef}^\sharp]$.

We use allocation-site abstraction [3] 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 can convey an intended meaning, while reading from an unassigned variable is generally an error. To distinguish between these two scenarios, our analysis uses two different abstract values for these two cases 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 $\mathsf{Int}^\sharp$) and specific values (such as constant strings), but also their combinations. 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 arrays (i.e. maps), for termination and efficiency reasons we chose to simplify them by computing them point-wise,

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

The set of union types forms a lattice where the partial order corresponds to the notion of subtyping. We denote type unions by the symbol $\cup$, which is the exact version of $\sqcup$. We define subtyping for unions by $\tau \sqsubseteq (\tau_1 \cup \tau_2) \iff \tau \sqsubseteq \tau_1 \vee \tau \sqsubseteq \tau_2$ and $(\tau_1 \cup \tau_2) \sqsubseteq \tau \iff \tau_1 \sqsubseteq \tau \wedge \tau_2 \sqsubseteq \tau$. We define the subtype relation 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 respectively the subtype and the supertype of all array types. Our approach relies on the fact that arrays in PHP, contrary to arrays in e.g. Java, are not objects and do not introduce a level of reference indirection.

3.1 Transfer Functions

For space reasons we only highlight less standard aspects of our abstract transfer functions. A compact description of the transfer functions of our analysis in Scala is given in around 1000 lines of Scala source code.[7]

Type refinement. Since the PHP language allows programs with little to no type annotations, it is often the case that types of values are completely unknown before they are used. 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 example, the code $\mathsf{\$b = \$a + 1; \$c = \$a + 2;}$ generates only one notice in PHANTM. Namely, after the first statement PHANTM assumes that $\mathsf{\$a}$ is a valid operand for mathematical operations, and refines its type to $\mathsf{Int}^\sharp \cup \mathsf{Float}^\sharp$. To achieve this in the general case, PHANTM computes the lattice meet between the type lattice elements corresponding to the current and the expected variable types. For example, a typical computation of the intersection of array types gives

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

Such type refinement corresponds to applying an **assume** statement that is a consequence of successful execution of an operation.

Conditional filtering. PHANTM also applies type refinement for assume statements implied by control structures. Note that PHP allows values of every type to be used as boolean conditions, and gives different boolean values to inhabitants of those types. This allows PHANTM to do refinement on the types of values used as boolean conditions. For example, the type **null** can only evaluate to false, whereas integers may evaluate to either true or false (true unless the value is $\mathsf{0}$). This is especially useful for booleans, for which we also define **true** and **false** as types. We can precisely annotate a function returning false on error, and a different type on successful execution. PHANTM can then use type refinement to verify code that invokes a function and checks for errors in the invocation. If the representation of the value becomes $\perp$ during the refinement, PHANTM concludes that the branch cannot be taken, detecting unreachable code.

Enforcing Termination. Given our allocation-side model for handling the heap, we identify two remaining potential sources of an infinite-height lattice: nested arrays and unions of singleton scalar types. For arrays, we limit the array nesting depth to a constant (five, in the current implementation). For singleton scalar types, we make sure that new singleton scalar types cannot be generated except when abstracting a literal or a run-time state value. Any operation handling singleton scalar types will either have one of them as a result type, or have a more general, widened type. We have found this approach to work well in practice (see Figure 4 for analysis performance).

[7] Please consult the file `src/phantm/types/TypeTransferFunction.scala` in the repository at `http://github.com/colder/phantm/`

3.2 Reporting Type Errors using Reconstructed Types

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

4 Runtime Instrumentation

Many PHP applications can be separated into two parts: the bootstrapping code and the core functionality of the application. The bootstrapping code is responsible for handling configuration settings, loading external libraries, loading sessions or including the appropriate definitions. Because this part of the code strongly depends on the configuration of the environment at hand, it usually cannot be analyzed statically. Compounding the problem of imprecision is that these configuration values that are approximated imprecisely tend to be used often in the rest of the code. To overcome this problem, Phantm includes a PHP library to instrument the analyzed application; one uses it to define a milestone in the code at which multiple aspects of the runtime state should get inspected. Using the state captured at this program point as the alternative starting point for static analysis, Phantm can use information that goes beyond the source code and produce an overall better output.

4.1 State Recovery

The runtime information that Phantm extracts includes: 1) all defined variables and constants and their associated values 2) a trace of function and class definitions, including the location in the code where the definition occurs, and 3) a trace of all included files. The library function used to mark the milestone and to collect the runtime information is called phantm_collect_state. It takes an array of variables as an argument, and is typically invoked as

```
phantm_collect_state(get_defined_vars());
```

When phantm_collect_state is called, the runtime state and the list of active definitions are stored into a file. This file can then be imported back into Phantm using the `--importState` option.

Phantm then loads the information and applies the following steps:

1. Attach the source of all included files to the AST of the main file.
2. Collect the function and class declarations that match the trace.
3. Create an abstract state s from the stored values for variables and constants.

4. Locate in the AST the program point p with the call to phantm_collect_state, and attach s to that program point.
5. Apply the static analysis starting from p.

In the reconstructed abstract state s, all scalar variables are associated to a singleton type that precisely describes the value from the collected state. A fresh, virtual, allocation site is associated to each object that was known at instrumentation time, and arrays are reconstructed with the correct set of keys and values (up to a bounded array nesting depth). The only limitation in practice is that resources cannot be reconstructed, because they typically cannot be serialized.

Summary of runtime analysis benefits. In our experience, runtime information improves the precision of our analyzer in the following ways:

Global variables. Projects like DokuWiki make an extensive use of global variables, e.g. to store configuration settings and database connections. Global variables are difficult to analyze in a purely static approach. Because they are typically defined in an initialization phase, our runtime instrumentation can capture their value; Phantm can then use it in the static analysis phase.

Increased and precise definition coverage. Phantm records the files that have been included during execution. Often all necessary libraries are included at the time of the phantm_collect_state indications, which means that all necessary functions are defined. When such dependencies are dynamic, they are not resolved with purely static analysis, resulting in warnings about undefined functions and results that are either useless or unsound (depending on whether missing the functions are assumed to perform arbitrary changes or no changes).

5 Evaluation

We evaluated Phantm on three PHP applications. The first one is an email client which we will call WebMail, similar in functionality to IMP.[8] 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 application is the popular open source wiki project DokuWiki and the third application is SimplePie, an open source library to manage the aggregation of RSS and Atom news feeds.

We first summarize the results of our evaluation without runtime instrumentation in Figure 4. "Warnings" is the number of warnings Phantm emitted with normal verbosity, while "Filtered Warnings" is using a special mode which focuses on most relevant errors. "Problems" is the number of problems identified, including actual bugs, dangerous implicit conversions, statements that could issue notices in PHP, and errors in annotations. We see that even for large code bases, the time required by the analysis remains reasonable.

In the sequel we show how runtime instrumentation helped improve these results. Finally, we describe a number of issues discovered with Phantm.

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

	Lines of code	Warnings	Filtered Warnings	Problems	Analysis Time
DokuWiki	31486	1232	270	76	244s
WebMail	3621	272	59	43	11s
SimplePie	15003	881	327	84	21s
Total	*50110*	*2385*	*656*	*203*	*276*s

Fig. 4. Summary of evaluation results without runtime instrumentation.

	Lines	Without	With	Δ	Reduction
updateprofile	62	19	0	19	100%
act_resendpwd	90	16	5	11	69%
check	143	14	4	10	71%
auth_ismanager	70	12	6	6	50%
auth_login	49	10	4	6	60%

Fig. 5. Effects of runtime instrumentation on DokuWiki. "Without" is the number of warnings emitted by Phantm without runtime instrumentation. "With" is the number of warnings emitted by Phantm with the information from runtime instrumentation about global variables and the type of arguments. In both cases, the function body is analyzed entirely.

We evaluated the impact of runtime instrumentation on DokuWiki and WebMail. The code of of both projects is structured as a loading phase followed by code that uses the configuration data. Consequently, the benefits of runtime information are considerable. We illustrate the impact of runtime information for DokuWiki in Figure 5, listing several functions among those for which runtime instrumentation brought the most significant improvement. Note that a comparison of the total number of warnings is not sensible, because using instrumentation can add code to the analysis that cannot be discovered statically.

Observe that we obtain a substantial reduction in the case, for example, of updateprofile. This is explained by the fact that this function primarily deals with global variables, user-provided form elements, and the current logged user, which is runtime-dependent. In essence, such functions illustrate the limitations of purely static analyses, and show how helpful runtime instrumentation was in overcoming these limitations.

Overall, for functions analyzed both with and without runtime information, 109 warnings (12%) were eliminated when using runtime information for DokuWiki and 18 (12%) for WebMail. Using the instrumentation had no notable impact on the analysis time; the overhead was only in the one-time loading of the saved state, which takes around one second in our implementation.

5.1 Issues Identified by Phantm

We now describe a small selection of issues in the three applications that we identified by inspecting the warnings emitted by Phantm.

WebMail bug 1) 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 indicates that **substr**() expects a string as its second argument, but that in this case the type False ∪ String was found. The developers were assuming that $charlist would always contain $char even though it was not always the case. Because of this bug, some of the passwords were improperly stored, potentially resulting in email accounts being inaccessible from WebMail and thus compromising WebMail's core functionality.

WebMail bug 2) 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 and could have led to new bugs as the code evolves further.

WebMail bug 3) 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 is accepted by the standard PHP interpreter even though $dist is undefined in the program; it contains a typographic error preventing the desired value from being stored in the array $dict.

WebMail bug 4) The tool identified several warnings for code such as $i = $str * 1, which casts a string into an integer using the implicit conversion triggered by the multiplication. Although it is not incorrect, it is flagged as bad style.

DokuWiki bug 1) 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" evaluates to true, however, "false" would evaluate to true as well!

DokuWiki bug 2) Keeping code documentation synchronized with the code itself is often problematic. As an illustration of this fact, Phantm uncovered over 25 errors in the existing annotations of arguments and return values.

DokuWiki bug 3) 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**, resulting in an infinite loop.

SimplePie bug 1) 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, which always results in **false**, and then computes the bitwise conjunction, while the goal is clearly to check whether a flag is set in $file→method. PHANTM finds the error by reporting that the right-hand side of **&** is a boolean value, and that an integer was expected.

SimplePie bug 2) PHANTM flags the following code as type incorrect:

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

An 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. As a result, the computation is incorrect when base64 is spelled with a capital "b", for instance.

6 Related Work

Data-flow analysis for type inference. Our work performs type inference using an abstract interpretation, resulting in a flow-sensitive static analysis. A systematic analysis of type analyses of different precision is presented in [4].

Static analysis of PHP. Existing work on statically analyzing PHP is primarily focused on the specific task of detecting security vulnerabilities and preventing attacks. PIXY [10] is a static analysis tool checking for vulnerabilities such as cross site scripting (XSS) or SQL injections, which remain the main attack vectors of PHP applications. Wassermann and Su [14] present work on statically detecting SQL injections using grammar-based specifications. Huang et al. [7] present a technique to conservatively prevent, rather than detect, similar attacks. They use a combination of code instrumentation, to automatically secure PHP scripts, and a static taint analysis, to reduce the number of additional checks. All these approaches focus on one analysis domain and make use of specific techniques and annotations. PHANTM on the other end ambitions to be useful in improving the quality of arbitrary PHP code and code documentation, while it can also serve to detect vulnerabilities, as illustrated in Section 5.1.

It is only recently that some work have been focusing on static analysis of types in PHP applications. Notably, the Facebook HIPHOP project[9] 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

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

native C++ type. In case such a type cannot be inferred, it simply falls back to a generic type.

The recently released tool PHPLint[10] aims to detect bugs through type errors. Even though its goal is close to the present work, Phantm has a much more precise abstract domain, and therefore reports many fewer spurious warnings. For instance, PHPLint fails to analyze precisely the initial example in Section 2 because it does not support arrays containing mixed types. Furthermore, it does not have union types, so many PHP functions will not be represented both soundly and precisely enough to 1) detect defects such as the Dokuwiki bug 3 of Section 5.1 and the **fopen** example of Section 2, while 2) avoiding false warnings when the developer correctly checks for return codes.

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 [5, 1]. It supports first class functions, but is not flow-sensitive and does not support value-array types.

In [11] researchers present an analysis of Cobol programs that recovers information corresponding to tagged unions. The work on the C programming language [9, 2] 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 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.

In [8] the authors present a type analysis for JavaScript also based on data-flow analysis. The abstract domain for array types presented in our paper goes beyond what is supported in [8]. On the other hand, the support for interprocedural analysis and pointer analysis in [8] is more precise than in the present paper. The main difference, however, is that we demonstrate the potential of combining dynamically computed program states with data-flow analysis.

Combining static and dynamic analysis. Combining static and dynamic analysis arises in a number of approaches. Our approach is closest to [12] and [15]. Promising approach have been developed that combine testing, abstraction, theorem proving [16] or combine may and must analysis [6]; these approaches compute a sound overapproximation, in contrast to our runtime information that performs a sample of an early stage of the execution to estimate properties of a dynamic environment.

7 Conclusion

Our experience with over 50000 lines of PHP code showed our tool to be fast enough and effective in identifying serious issues in code such as exploits, infinite

[10] `http://www.icosaedro.it/phplint/`

loops, and crashes. The use of runtime information was shown to be helpful in reducing the number of false alarms in the tool and focusing the attention on true errors. We therefore believe that it is well-worthwhile to build into future static analyses tools the ability to start the analysis from a recorded concrete program state. This approach overcomes several limitations of purely static approach while preserving certain predictability that help interpret the results that it computes. Our tool Phantm is available for download and evaluation, and we report verifiable experimental results on significant code bases, including popular software whose source code is publicly available.

References

1. Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL*, 1994.
2. Satish Chandra and Thomas Reps. Physical type checking for C. In *Workshop on Program analysis for software tools and engineering (PASTE)*, 1999.
3. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.
4. Patrick Cousot. Types as abstract interpretations. In *POPL*, 1997.
5. Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, 1992.
6. Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, 2010.
7. Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, 2004.
8. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, 2009.
9. Ranjit Jhala, Rupak Majumdar, and Ru-Gang Xu. State of the union: Type inference via Craig interpolation. In *TACAS*, pages 553–567, 2007.
10. Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symp. Security and Privacy*, 2006.
11. Raghavan Komondoor, Ganesan Ramalingam, Satish Chandra, and John Field. Dependent types for program understanding. In *TACAS*, 2005.
12. Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.
13. Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.
14. Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.
15. Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Transactions on Computer Systems*, 28(1), 2010.
16. Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA*, pages 145–156, 2006.