WeVerca: Web Applications Verification for PHP (Tool Paper)*

David Hauzar and Jan Kofroň

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Abstract. Static analysis of web applications developed in dynamic languages is a challenging yet very important task. In this paper, we present WeVerca, a framework that allows one to define static analyses of PHP applications. It supports dynamic type system, dynamic method calls, dynamic data structures, etc. These common features of dynamic languages cause implementation of static analyses to be either imprecise or overly complex. Our framework addresses this problem by defining end-user static analyses independently of value and heap analyses necessary just to resolve these features. As our results show, taint analysis defined using the framework found more real problems and reduced the number of false positives comparing to existing state-of-the-art analysis tools for PHP.

1 Introduction

PHP is the most common programming language used at the server side of web applications. It is notably used, e.g., by Wikipedia and Facebook. PHP as well as other dynamic languages contains dynamic features, such as dynamic type system, dynamic method calls (names of called methods are computed at run-time), and dynamic data structures (names of object fields are computed at run-time and object fields can be added at run-time). These features provide flexibility accelerating the development. However, they make applications more error-prone and less efficient. Consequently, they shift more work to tools for error detection, code refactoring, and code optimization.

For most of these tools, static program analysis is a necessary prerequisite. Unfortunately, dynamic features pose major challenges here. To precisely resolve these features, the end-user analysis (e.g., taint analysis) needs to be combined with value and heap analyses. Importantly, these analyses must interplay. To resolve dynamic accesses to data structures, the heap analysis needs to evaluate value expressions and the value analysis must track values over heap elements—array indices and object fields.

In this paper we present WeVerca[1], an open-source static analysis framework for PHP. WeVerca allows to define end-user static analyses independently

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by Charles University institutional funding SVV-2014-260100.

[1] `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`

of dynamic features. This is possible because: (1) WeVerca defines an interplay of value and heap analyses allowing to define these analyses independently of each other. (2) WeVerca comes with default implementations of context-sensitive heap and value analyses that model associative arrays and prototype objects, track values of PHP primitive types, and model library functions, native operators, and type conversions. (3) WeVerca defines how information from heap and value analyses are used to resolve dynamic features (i.e., to compute control-flow and resolve dynamic data accesses). As a proof of the concept, we implemented static taint analysis for detection of security problems.

2 Example

As an example, consider static taint analysis, which is commonly used for web applications. It can be used for detection of security problems, e.g., SQL injection and cross-site scripting attacks. The program point that reads user-input, session ids, cookies, or any other data that can be manipulated by a potential attacker is called *source*, while a program point that prints out data, queries a database, etc. is referred to as *sink*. Data at a given program point are *tainted* if they can pass from a source to this program point. A tainted data are *sanitized* if they are processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potential malicious parts of it. Program is *vulnerable* if it contains a sink that uses data that are tainted and not sanitized.

Static taint analysis can be performed by computing the propagation of tainted data and then checking whether tainted data can reach a sink. The propagation of tainted data computed by forward data-flow analysis is shown in Tab. 1[2]. The analysis is specified by giving the lattice of data-flow facts, the initial values of variables, the transfer function, and the join operator.

Lattice	L	$true$	
Top	$\top$	Bool	
Initial value	$init(v)$	$true$	if $v \in$ \$_SESSION $\cup$ ..
		$false$	otherwise
Transfer function	$TF(LHS = RHS)$	$var = \bigvee_{r \in RHS} r$	if $var \in LHS$
		$var = var$	otherwise
	$TF(n)$	$var = var$	if n is not assignment
Join operator	$\sqcup(x, y)$	$x \vee y$	

Table 1. Propagation of tainted data.

Consider now the code in Fig. 1. At lines (1)–(9) classes for processing the output are defined. They can either log the output or show the output to the user. While the `Templ1` class uses a *sink* command to show the output, `Templ2` uses a *non-sink* command (e.g., does not send the output to the browser directly, but sanitizes it first). At lines (13)–(16) the application mode is set based on the value of `DEBUG` either to `log`—the application will log the output—or to `show`—the application will show the output to the user. At lines (17)–(20) the skin is set based on user input. At line (21), the array `$users` is initialized with

[2] For simplicity we omit the specification of sanitization.

the address of administrator. This value is not taken from any source and can be directly shown to the user. Note the update at line (11) is correct even if the variable `$users` is uninitialized. In PHP, if a non existing index is updated, it is automatically created and if the update involves next dimension, the index is initialized with an empty array. At lines (23)–(24) information about the user name and user address is assigned to the array `$users`. Note that this information is tainted. Finally, at lines (25)–(26) data are processed to the output.

```
1  class Templ {
2    function log($msg) {...}
3  }
4  class Templ1 : Templ {
5    function show($msg) { sink($msg); }
6  }
7  class Templ2 : Templ {
8    function show($msg) { not_sink($msg); }
9  }
10 function initialize(&$users) {
11   $users['admin']['addr'] =
           get_admin_addr_from_db();
12 }
13 switch (DEBUG) {
14   case true: $mode = "log"; break;
15   default: $mode = "show";
16 }
17 switch ($_GET['skin']) {
18   case 'skin1': $t = new Templ1(); break;
19   default: $t = new Templ2();
20 }
21 initialize($users);
22 $id = $_GET['userId'];
23 $users[$id]['name'] = $_GET['name'];
24 $users[$id]['addr'] = $_GET['addr'];
25 $t->$mode($users[$id]['name']);
26 $t->$mode($users['admin']['addr']);
```

Fig. 1. Running example

The code contains two vulnerabilities. At lines (25) and (26) the method `show` of `Templ1` can be called, its parameter `$msg` can be tainted and the parameter goes to the sink. Taint analysis defined using WeVerca detects both vulnerabilities. Note that the definition of taint propagation uses just the information in Tab 1. This is possible only because WeVerca automatically resolves control-flow and accesses to built-in data structures. That is, WeVerca computes that the variable `$t` can point to objects of types `Templ1` and `Templ2` and that the variable `$mode` can contain values `show` and `log`. Based on this information, it resolves calls at lines (25) and (26). Moreover, as WeVerca automatically reads the data from and updates the data to associative arrays and objects, at line (24), the tainted data are automatically propagated to index `$users['admin']['addr']` defined at line (11). Consequently, the access of this index at line (26) reads tainted data.

3 Tool description

The architecture of WeVerca is shown in Fig. 2. For parsing PHP sources and providing abstract syntax tree (AST) WeVerca uses Phalanger[3]. The analysis is split into two phases. In the first phase, the framework computes control-flow of the analyzed program together with the shape of the heap and information about values of variables, array indices and object fields. Then it also evaluates expressions used for accessing data. The control-flow is captured in the intermediate representation (IR), while the other information is stored in the data representation. IR defines the order of instructions' execution and has function calls, method calls, includes, and exceptions already resolved. In the

[3] `http://www.php-compiler.net/`

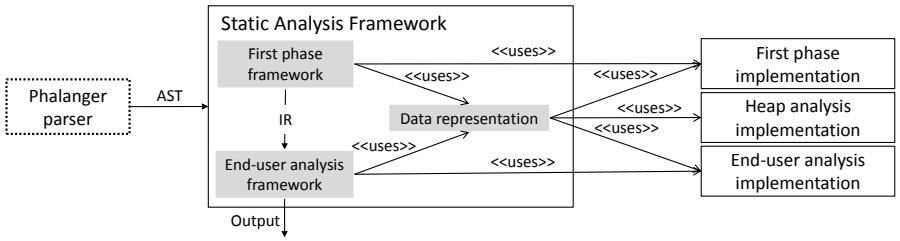

Fig. 2. The architecture of WeVerca

second phase, end-user analyses of the constructed IR are performed. The tool includes the following parts:

- **Data Representation** stores analysis states and allows to access them—it allows to read values from data structures, write values to data structures, and modify the shape of data structures. Next, it performs join and widening of the states and defines their partial order. Importantly, data representation defines the interplay of heap and value analyses allowing each analysis to define these operations independently. WeVerca contains implementation of heap analysis described in [1]. It supports associative arrays and objects of an arbitrary depth (in PHP, updates create indices and properties if they do not exist and initialize them with empty arrays and empty objects if needed; on contrary, read accesses do not, so updates of such structures cannot be decomposed). Accesses to these structures can be made using an arbitrary expression yielding even statically unknown values.
- **First-phase implementation** must define value analysis that tracks values of PHP primitive types and evaluates value expressions. Next, it must handle declaration of functions, classes, and constants. Finally, it must compute targets of include statements and function and method calls, and it must define context sensitivity. WeVerca contains a default implementation of the first phase providing fully context-sensitive value analysis precisely modeling native operators, native functions, and implicit conversions.
- **End-user analyses** can be specified using an arbitrary value domain. This is possible because (1) control-flow is already computed, (2) the shape of the heap is computed and dynamic data accesses are resolved—all information that data representation needs to discover accessed variables, indices, and fields are available. (3) Data representation combines heap and value analyses automatically, i.e., to perform operations with analysis states, it uses standard operations of combined analyses. The framework contains an implementation of static taint analysis as a proof-of-the-concept.

4 Results

To evaluate the precision and scalability of the framework, we used the framework to implement static taint analysis and we applied it to a NOCC webmail client[4] and a benchmark application comprising of a fragment of the myBlog-

[4] http://nocc.sourceforge.net/

gie weblog system[5], with a total of over 16,000 lines of PHP code. While the benchmark application contains 13 security problems, in the case of the webmail client, the number of problems is not known.

Tab. 2 shows the summary of results together with the results of Pixy [3] and Phantm [4], the state-of-the-art tools for security analysis and error discovery in PHP applications. The table shows that the analysis defined using WeVerca outperforms the other tools both in error coverage and number of false positives when analyzing the benchmark application. While it took WeVerca more than 5 minutes to analyze the webmail client and 52 alarms were reported, Pixy was even not able to analyze this application. Phantm analyzed the application in two minutes, however, the false-positive rate of 93% makes its output almost useless.

Out of 13 problems in the benchmark application, WeVerca discovered all of them. One of the false alarms reported by WeVerca is caused by imprecise modeling of the built-in function `date`. WeVerca only models this function by types and deduced that any string value can be returned by this function. However, while the first argument of the function is `"F"`, the function returns only strings corresponding to English names of months. When the value returned by this function is used to access the index of an array, WeVerca incorrectly reports that an undefined index of the array can be accessed. Two remaining false alarms are caused by path-insensitivity of the analysis. The sanitization and sink commands are guarded by the same condition, however, there is a joint point between these conditions, which discards the effect of sanitization from the perspective of path-insensitive analysis. While the first false-alarm can be easily resolved by modeling the built-in function more precisely, the remaining false alarms would require more work. One can either implement an appropriate relational abstract domain or devise a method of path-sensitive validation of alarms.

	Lines	**WeVerca** W/C/F/T	**Pixy** W/C/F/T	**Phantm** W/C/F/T
myBloggie	648	16/**100**/**19**/2.2	16/69/44/**0.6**	43/23/93/2.5
NOCC 1.9.4	15605	52/NA/NA/332	NA	426/NA/NA/**130**

Table 2. Comparison of tools for static analysis of PHP. W/C/F/T: **W**arnings / error **C**overage (in %) / **F**alse-positives rate (in %) / analysis **T**ime (in s). The best results are in bold.

5 Related work

The existing work on static analysis of PHP and other dynamic languages is primarily focused on specific security vulnerabilities and type analysis.

Pixy [3] performs taint analysis of PHP programs and it provides information about the flow of tainted data using dependence graphs. It involves a flow-sensitive, interprocedural, and context-sensitive data flow analysis along with

[5] http://mybloggie.mywebland.com/

literal and alias analysis to achieve precise results. The main limitations of Pixy include limited support for statically-unknown updates to associative arrays, ignoring classes and the `eval` command, omitting type inference, and limited support for handling file inclusion and aliasing. Alias analysis introduced in Pixy incorrectly models aliasing when associative arrays and objects are involved.

Phantm [4] is a PHP 5 static analyzer for type mismatch based on data-flow analysis; it aims at detection of type errors. To obtain precise results, Phantm is flow-sensitive, i.e., it is able to handle situations when a single variable can be of different types depending on program location. However, they omit updates of associative arrays and objects with statically-unknown values and aliasing, which can lead to both missing errors and reporting false positives.

TAJS [2] is a JavaScript static program analysis infrastructure. To gain precise results, it models prototype objects and associative arrays, dynamic accesses to these data structures, and implicit conversions. However, TAJS combines combines heap and value analysis ad-hoc, which results in intricate lattice structure and transfer functions.

6 Conclusion and future work

In this paper, we presented WeVerca, a framework for static analysis of PHP applications. WeVerca makes it possible to define static analyses independently of dynamic features, such as dynamic includes, dynamic method calls, and dynamic data accesses to associative arrays and objects. These features are automatically resolved using information from heap and value analyses, which are automatically combined.

Our prototype implementation of static taint analysis outperforms state-of-the-art tools for analysis of PHP applications both in error coverage and the false-positive rate. We believe that WeVerca can accelerate both the development of end-user static analysis tools and the research of static analysis of PHP and dynamic languages in general.

For future work, we plan to improve the scalability and precision of analyses provided by the framework. In particular, this includes the scalability improvements of data representation, implementation of more choices of context-sensitivity, more precise widening operators, and devising precise modeling of more library functions.

References

1. D. Hauzar, J. Kofroň, and P. Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS'14*, EPTCS, 2014.
2. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS'09*. Springer-Verlag, 2009.
3. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *S&P'06*. IEEE, 2006.
4. E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP Analyzer for Type Mismatch. In *FSE'10*. ACM, 2010.