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 features such as dynamic type system, virtual method calls, and dynamic data structures. These features are common in PHP and other dynamic languages, unfortunately, they cause implementation of static analysis to be either imprecise or overly complex. Our framework addresses this problem by providing options to define end-user static analyses independently of other 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 features such as dynamic type system, virtual methods, reflection, and dynamic data structures. These features provide flexibility and accelerate the development, in particular, the development of web applications. However, they make these applications more error-prone and less efficient. Consequently, they shift more work to tools for security analysis, error detection, code refactoring, code optimization, and code navigation.

For most of these code analysis tools, static program analysis is a necessary prerequisite. Unfortunately, dynamic features pose major challenges here. To resolve these features, the end-user analysis (e.g. taint analysis) needs to be combined with other analyses, e.g., type analysis, literal analysis, and points-to analysis. Moreover, there are many choices of implementing these supporting analyses that affect the scalability and precision of the resulting tool, e.g., the choice of context sensitivity, the choice of abstract domains, and the way in which library functions are modeled [3]. This makes implementation of any end-user static analysis either complex or imprecise.

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S.

In this paper we present WeVerca[1], an open-source static analysis framework for PHP. WeVerca makes it possible to define end-user analyses independently on dynamic constructs by automatically resolving the constructs using the supporting analyses.

WeVerca comes with default implementations of supporting analyses. They include inter-procedural context-sensitive type, literal, and points-to analyses. They employ a set and interval abstract interpretation domains, model built-in data structures such as associative arrays and objects, library functions, native operators, and type conversions. Next, 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 which 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 it is processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potential malicious parts of it. An execution of a PHP program is called *vulnerability* if data can pass from a source to a sink without being 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

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

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

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 `$users` is initialized with the address of administrator. This value is not taken from any source and can be directly shown to the user. Note that in PHP, if a value is assigned to an index and this index does not exist, it is automatically created. Next, at lines (23)–(24) information about the user name and user address is assigned to the array `$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 `$message` can be tainted and the parameter goes to the sink. Taint analysis defined using WeVerca detects both vulnerabilities. Note that the definition of taint propagation uses just the information in Tab 1. This is possible only because WeVerca automatically resolves control-flow and accesses to built-in data structures. That is, WeVerca computes that the variable `$t` can point to objects of types `Templ1` and `Templ2` and that the variable `$action` can contain values `show` and `log`. Based on this information, it automatically resolves calls at lines (25) and (26). Moreover, as WeVerca automatically reads the data from and updates the data to associative arrays and objects, at line (24), the tainted data are automatically propagated to index `$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 [4]. WeVerca takes an PHP program as input and analyses it. The computation is split into two phases. In the first phase, WeVerca constructs the control-flow of the analysed program together with the shape of the heap, information about the values, types of variables, array indices and object properties. 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.

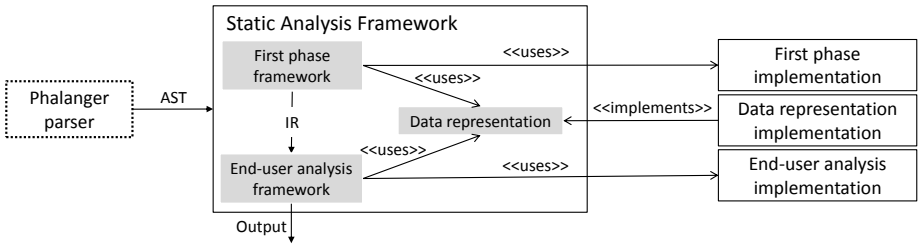

Fig. 2. The architecture of WeVerca

In the second phase, end-user analyses of the constructed IR are performed. WeVerca provides a simple API for manipulation with data in the second phase, e.g. in assignments, data are automatically propagated to all possible targets. The tool includes the following parts:

- **Data Representation** defines how data are read from and written to data structures. The implemented data representation supports associative arrays as well as objects of an arbitrary depth (in dynamic languages, updates create indices and object fields if they do not exist, read accesses do not, consequently, updates of such structures cannot be decomposed to updates of depth one). Accesses to these structures can be made using an arbitrary expression at any level.
- **First-phase Analyses** must provide information necessary for computing control-flow of the program and accessing data. WeVerca defines how these analyses interplay and also provides a default implementation of these analyses. The provided analyses are inter-procedural, context sensitive, and use the set and interval abstract domains. To guarantee convergence of the analyses, widening can be applied on these domains. The analyses enhance the precision by refining the data for the assume statements implied by control structures. They precisely model all native operators, type conversion, and selected library functions and other library functions by types.
- **Implementation of End-user Analysis.** 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 implemented static taint analysis using the WeVerca framework and we applied it to a NOCC webmail client[3] and a benchmark application comprising of a fragment of the myBloggie weblog system[4], 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 [1] and Phantm [2], the state-of-the-art tools for security analysis and error discovery

[3] http://nocc.sourceforge.net/
[4] http://mybloggie.mywebland.com/

in PHP applications. The table shows that the analysis defined using WeVerca outperfroms the other tools both in error coverage and number of false positives when analysing the benchmark application. While it took WeVerca more than 13 minutes to analyze the webmail client and more than 100 alarms were reported, Pixy was even not able to analyze this application. Phantm analyzed the application in two minutes, however, it reported more than 19 cases where it was not able to resolve an include statement indicating that it did not analyze the whole application.

Out of 13 problems in the benchmark application, WeVerca discovers 10 of them. Two problems that were not discovered comprise of accessing potentially undefined variable when evaluating the expression in the condition and the remaining problem comprises of making a potentially undefined global variable accessible in the local scope.

One false alarm reported by WeVerca is caused by imprecise modeling of 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	13/**78**/**23**/2.6	8/31/50/**0.6**	43/46/86/2.5
NOCC 1.9.4	15605	101/NA/NA/817	NA	426/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 [1] is an open-source tool for detection of taint-style vulnerabilities in PHP 4. It involves a flow-sensitive, interprocedural, and context-sensitive data-flow analysis along with literal and alias analysis to achieve precise results. The main limitations of Pixy include a limited support for statically-unknown updates to associative arrays, ignoring classes and the `eval` command, and limited support for aliasing and handling file inclusion, which all represent principle dif-

ferences from programming languages such as Java and C. Importantly, Pixy does not perform type inference, which also limits its precision and soundness.

Stranger [5] is an automata-based string analysis tool for PHP, which is built upon Pixy. It adds a more precise string manipulation techniques that enable the tool to prove that an application is free from attack patterns specified as regular expressions. However, it shares the other limitations of Pixy.

Phantm [2] is a PHP 5 static analyzer based on data-flow analysis aiming at detection of type errors. It combines run-time information from the bootstrapping phase of an application and static analysis when instrumentation using this information is used. 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. However, it omits updates of associative arrays and objects with statically-unknown values and aliasing, which can lead to both missing errors and reporting false positives.

6 Conclusion and future work

In this paper, we presented WeVerca, a framework for static analysis of PHP applications. WeVerca specifies which analyses are necessary for computing control-flow of the program and accessing data, supporting associative arrays and objects, and defines how these analyses interplay. It also provides default implementations of these analyses. Next, WeVerca allows to define end-user analyses, while automatically resolving control-flow and data accesses. 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 library functions.

References

1. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *S&P'06*. IEEE, 2006.
2. E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP Analyzer for Type Mismatch. In *FSE'10*, 2010.
3. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
4. Phalanger. `http://phalanger.codeplex.com/`.
5. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. *TACAS'10*, 2010.