Data-flow Analysis of Programs with Associative Arrays*

David Hauzar, Jan Kofroň, and Pavel Baštecký
Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University in Prague
Czech Republic
{hauzar, kofron}@d3s.mff.cuni.cz, anebril@seznam.cz

Dynamic programming languages, such as PHP, JavaScript, and Python, provide built-in data structures including associative arrays and objects with similar semantics—object properties can be created at run-time and accessed via arbitrary expressions. While a high level of security and safety of applications written in these languages can be of a particular importance (consider a web application storing sensitive data and providing its functionality worldwide), dynamic data structures pose significant challenges for data-flow analysis making traditional static verification methods both unsound and imprecise. In this paper, we propose a sound and precise approach for value and points-to analysis of programs with associative arrays-like data structures, upon which data-flow analyses can be built. We implemented our approach in a web-application domain—in an analyzer of PHP code.

1 Introduction

Dynamic languages are often used for the development of security-critical applications, in particular web applications. To assure a reasonable level of reliability, various static analyses such as security analysis and bug finding are applied. In these cases, data-flow analysis is a necessary prerequisite. Unfortunately, dynamic features pose major challenges here. For instance, any interprocedural data-flow analysis needs to track types of variables to determine targets of virtual method calls. This becomes even more important in the case of languages with dynamic type systems, where types of variables can be completely unspecified. Moreover, names of methods to be called or files to be included can be computed at run-time. Next, all these data can be manipulated using built-in dynamic data structures, such as multi-dimensional associative arrays and objects with similar semantics—object properties can be created at run-time and accessed via arbitrary expressions, e.g., variables. This happens relatively often, e.g., in web applications, which manipulate a lot of input.

Tracking values of variables and analysing the shape of these data structures is thus essential for data-flow analysis of dynamic languages. In this paper we present our approach to these challenges. Our contribution includes: (1) value and points-to analysis of associative arrays with an arbitrary depth and accessible using statically unknown values and arbitrary expressions, (2) modeling explicit aliases, and (3) a prototype implementation.

2 Motivation and Overview

In this section, we show some dynamic features that impact the data-flow analysis and present an overview of our approach. We use PHP as the representative of a dynamic language; However, other languages, especially those connected with the development of web applications, provide built-in associative arrays-like data structures. These languages includes Java-script, Python, Ruby, etc. Moreover, libraries of "ordinary" programming languages emulate some of these features and offer the developer API behaving in a similar way. Hence, our approach is not limited to PHP. For the illustration of our concepts, we use the code in Fig. 1 as a running example.

*This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by Charles University Foundation grant 431011 and by Charles University institutional funding SVV-2014-260100.

J. Pang and Y. Liu (Eds.): 3rd International Workshop
on Engineering Safety and Security Systems 2014 (ESSS 2014)
EPTCS 150, 2014, pp. 56–70, doi:10.4204/EPTCS.150.6

```
1  $any = $_GET['user_input']; // an arbitrary user input
2  $alias = 1; $alias2 = 1; $alias3 = 1;
3  if ($any) {
4          $arr[$any] = &$alias;
5          $t = $arr[1]; // t can be either undefined or can
               have value 1
6          $t[2] = 2; // can update also $alias[2] and e.g.
               $arr[1][2]
7          $arr[1][2] = 3;
8          $arr[1][3] = 4;
9          $arr[2][3] = 5;
10 } else {
11         $arr[$any][2] = 6;
12         $arr[1][$any] = 7; // can update also some of
               variables involved by the previous update
13 }
14
15 $arr[2][1] = &$alias2; // $arr[2][1] and $alias2 can be
       aliased also with $alias[1]
16 $arr[2] = &$alias3;
17 $arr2 = $arr; // deep-copies $arr, including aliases
18 $arr2[2] = 8; // updates also $arr[2] and $alias3
19 $arr2[3] = 9; // can update also $arr[3] and $alias
20 $arr[$any] = arr2;
```

Figure 1: Running example.

2.1 Variables, Arrays, and Objects

Variables as well as indices and object properties need not be declared. If a specified index exists in an array, it is overwritten; if not, it is created. At line 7 in Fig. 1, a new array is created in `$arr` and index `2` is added to this array. Next, at line 8, index `3` is added to this array.

Arrays can have an arbitrary depth. Unfortunately, updates of such structures cannot be decomposed. That is, splitting the update at line 7 into two updates at lines 5-6 results in different semantics. The first reason is that the array assignment statement deep-copies the operand. The update at line 6 thus does not update the array stored at `$arr[1]`, but its copy. The second reason is that while updates create indices if they do not exist, read accesses do not; while the update at line 7 creates an index containing an array in `$arr[1]` in the case it does not exist, the read access at line 5 returns `null` in this case and the update at line 6 fails.

The semantics of the PHP object model is similar to the semantics of associative arrays. Objects' properties need not to be declared. If a non-existing property is written, it is created. As well as indices, properties can be accessed via arbitrary expressions. Objects can also have an arbitrary depth in the sense of reference chains. In the following, we describe associative arrays, however, the same principles apply to objects as well. We write associative arrays-like data structures to emphasize this fact.

2.2 Dynamic Accesses

In dynamic languages, variables, indices of arrays, and properties of objects can be accessed with arbitrary expressions. At line 4 in Fig. 1 the `$arr` array with an index determined by the variable `$any` is assigned; if a given index exists in `$arr`, it is overwritten; if not, it is created. Therefore, the set of variables, array indices and object fields is not evident from the code.

An update can involve more than one element and can be statically unknown. The update at line 4 is statically unknown and thus may or may not influence accesses at lines 5, 7, 8, 9, 15, and 20. Similarly, line 11 can access index `2` in any index at the first level. In particular, it can access also index `1` at the first level, which is updated at the following line. That is, reading `$arr[1][2]` can return either of values `6`, `7`, and `undefined`, reading `$arr[1][1]` can return `7` and `undefined`, reading `$arr[2][2]` can return `6` and `undefined`, and reading `$arr[2][1]` always returns `undefined`. Next, after two branches of the if statement are merged at line 13, reading of `$arr[1][2]` can return values `6`, `7`, `3`, and `undefined`.

2.3 Explicit Aliasing

PHP makes it possible for a variable, index of an array, and property of an object to be an alias of another variable, index, or property. After an update of an element, all its aliases are also updated. Aliasing in PHP is thus similar to references in C++ in many aspects.

Unlike C++, in PHP each variable, index, and property can be aliased and later un-aliased from its previous aliases and become an alias of a new element. As an example, the statement at line 16 un-aliases

`$arr[2]` from its previous aliases. Moreover, a variable can be an alias of another variable only at some paths to a given program point, e.g., if it is made an alias in a single branch of an if statement.

The statement at line 4 makes variable `$alias` an alias of a statically unknown index of array `$arr`. Hence, the statement at line 7 accesses `$arr[1][2]` and may also access `$alias[2]`. Similarly, the statement at line 15 makes `$alias2` an alias of `$arr[2][1]` and may also make it an alias of `$alias[1]`. If an array is assigned, it is deep-copied. However, if an index in the source array has aliases, the set of aliases in the corresponding index in a target array consists of these aliases and the source index. Consequently, the statement at line 18 updates also `$arr[2]` and its alias `$alias3`. Similarly, the statement at line 19 may update also `$arr[3]` and `$alias`, because the statement at line 3 may make these aliases of each other.

2.4 Overview of the Approach

Our approach consists of the following key parts: (1) definition of analysis state, (2) definition of read accesses to associative arrays, (3) definition of write accesses to associative arrays (i.e. the transfer function), and (4) definition of merging associative arrays (i.e. the join operator).

The fundamental part of analysis state consists of represenation of associative arrays. Each associative array contains a set of indices including a special index called *unknown field*. This index stores information that has been written to statically unknown indices of the array. All indices (including unknown fields) can contain a set of values and also can point to another associative array—the next dimension. Next, unknown fields always contain value `undefined`. Note that (multi-dimensional) unknown fields allow for dealing with statically unknown accesses, however, they pose a challenge both to definition of new indices and definition of merging associative arrays.

Both indices that are read by a read access and indices that are updated by a write access to a multi-dimensional associative array are specified using a list of expressions. Each expression specifies an access to one dimension of the array. At each level, indices corresponding to values of an expression corresponding to the level are followed. If the expression yields a statically unknown value, all indices (including the unknown field) are followed. The read access differs from the write access in the way it handles the case when there is no corresponding index defined. While the read access follows the unknown field, the write access defines the index. Note that in the latter case, all data that could have been assigned to the new index using statically unknown updates are copied to this index. The reason is that these data are stored in unknown fields and unknown fields are not followed by statically known read accesses. Next, unlike a read access, a write access distinguishes indices that certainly must be updated and indices that only may be updated. Former indices are strongly updated—original data are replaced with new data, the latter indices are weakly updated—original data are joined with new data.

Example 1: *At line 5, value `1` is used to read-access the first dimension of an associative array. Because the array has no index corresponding to this value, the unknown field is followed. It contains value `1` from the update at line 4 and also value `undefined`.*

Example 2: *At line 12, value `1` is used to write-access the first dimension of an array `$arr`. The array has no index corresponding to this value and the index `$arr[1]` is thus created. Because statically unknown assignment at line 11 could involve also the index `$arr[1]`, the data from this assignment are copied to a new index and thus also the index `$arr[1][2]` (with values `6` and `undefined`) is defined.*

The principal challenge of merging multi-dimensional associative arrays with unknown fields is to determine the set of indices of the resulting array. That is, the resulting array may contain indices that are not present in any array being merged. The reason again stems from the fact that unknown fields are not followed by statically known read accesses.

Example 3: *As an example, see the join point at line 13. The array `$arr` contains all indices that are defined in either of merged branches plus the index `$arr[2][2]` (with values `6` and `undefined`).*

$s \in \Sigma$	$= Var \times Map \times Index \times Aliases$
$m \in Map$	$= Var \to \mathscr{P}(Val)$
$i \in Index$	$= (Var \times Val) \to Var$
$a \in Aliases$	$= Aliases_{must} \times Aliases_{may}$
$a_{must} \in Aliases_{must}$	$= Var \times Var$
$a_{may} \in Aliases_{may}$	$= Var \times Var$

Table 1: Data-flow analysis state-space.

$undefVar \in Var$	is a variable representing an undefined variable.
$* \in Val$	is the statically-unknown value.
$undefined \in Val$	is the undefined value.
$\bullet \in Val$	is the value representing index-name of *unknown* field.

Table 2: Special variables and values.

The reason of creating new index is that while in the second branch the read access to this index in the first level follows the unknown field and then reaches the value assigned at line 11, in the merged array the read access follows the index `$arr[2]`.

3 Formalization

We formalized our data-flow analysis using data-flow equations for the forward data-flow analysis [7]. The formalization includes handling associative arrays of unlimited depth and accesses with arbitrary expressions to such structures. It does not explicitly include handling of objects. While objects are treated analogously to arrays in our implementation, there are subtle differences. Therefore, we excluded handling of objects from our formalization to make it more clear.

3.1 Analysis State Space

Tab. 1 presents elements of the state space of our data-flow analysis. Every state contains a variable, which represents the symbol-table. Because top-level variables can be accessed dynamically (`$$var` is a variable whose name is given by a value of variable `$var`), we model top-level variables as indices of the symbol-table variable[1]. Function *Map* maps a variable to a set of its possible values. Function *Index* maps a variable and an index name to a variable containing an array which has the first variable on the index with this name. In the following, we say that variable v is an index of variable p identified by value *ind* if $((v, ind), p) \in indexOf$ (i.e., v is p["*ind*"]). A pair of relations *Aliases* relates variables that are *must* and *may* aliases. Tab. 2 presents special variables and values. The value $\bullet$ identifies the *unknown* field of a given variable. The *unknown* field of a variable is used to access statically-unknown indices of the variable. In Tab. 3, there are several helper functions (projections) defined; we use them in the subsequent definitions.

3.2 Data-flow equations

For propagating states through the nodes of the control-flow graph (CFG), we use a modification of standard data-flow equations for the forward problem. Each node k of CFG has six states associated: IN_k, GEN'_k, IN'_k, GEN_k, $KILL_k$ and OUT_k. IN_k represents the data coming to k; it is created by merging the states going out from all predecessors of k. In the case that the node has more predecessors, we call this state *join point* and the operation *mergeStates* merges information from different states. If the node has only one predecessor, the operation *mergeStates* only copies the information. The state GEN'_k defines

[1] Consequently, the notions of index and variable refer to the same abstraction and we use them interchangeably. That is, an index of an associative array in an arbitrary depth is a first class name the same way as a variable.

variables that are newly defined by the update and data of these variables (note that these data can come only from unknown fields). The state $KILL_k$ defines data that is removed from variables by the update while GEN_k defines data that is added from the right-hand side of the update statement. Finally, the state OUT_k represents updated data, i.e., the data going out from this node. The predicate $pred(k)$ returns the set of n output states associated with the predecessors of k.

The data-flow equations are:

$$\begin{aligned} IN_k &= mergeStates(\{OUT_p\}), p \in pred(k) \\ IN'_k &= IN_k \cup GEN'_k \\ OUT_k &= GEN_k \cup (IN'_k - KILL_k) \end{aligned}$$

For the the initial node i, the output state is as follows:

$$OUT_i = (root, \{(root, \emptyset), (unk, \{undefined\})\}, \{((unk, \bullet), root)\}, (\emptyset, \emptyset))$$

That is, the state contains variable *root* representing a symbol table and a variable *unk* which is its unknown field—it represents statically-unknown variables.

#	Expression	
	Let $x = (root, map, i, (a_{may}, a_{must}))$ be a state.	
(1)	$r(x)$	$= root$
(2)	$values(x, v \in Var)$	$= \{vals, (v \to vals) \in map\}$
(3)	$values(x, V \in \mathscr{P}(Var))$	$= \{values(x, v), v \in V\}$
(4)	$values_{undef}(x, V \in \mathscr{P}(Var))$	$= values(x, V)$ if $V \neq \emptyset$
		$= \{undefined\}$ if $V = \emptyset$
(5)	$values(x, undefVar)$	$= \{undefined\}$
(6)	$indexOf(x)$	$= i$
(7)	$indices(x, V \in \mathscr{P}(Val))$	$= \{iv, \exists_{v \in V} \exists_{n \in Val}((iv, n), v) \in i\}$
(8)	$indices(x, V \in \mathscr{P}(Var), I \in \mathscr{P}(Val))$	$= \{iv, \exists_{v \in V} \exists_{ind \in I}((iv, ind), v) \in i\}$
(9)	$aliases_{must/may}(x)$	$= \{(v_1, v_2), (v_1, v_2) \in a_{must/may} \vee (v_2, v_1) \in a_{must/may}\}$
(10)	$aliases_{must/may}(x, v \in Val)$	$= \{a, (v, a) \in aliases_{must/may}(x)\}$
(11)	$aliases_{must/may}(x, V \in \mathscr{P}(Val))$	$= \{(a, v), a \in aliases_{must/may}(x, v), v \in V\}$
(12)	$aliases(x, v \in Val)$	$= aliases_{may}(x, v) \cup aliases_{must}(x, v)$

Table 3: List of helper functions and projections.

3.3 Access Paths

We describe expressions for accessing variables and associative arrays of an arbitrary depth using access paths. An access path consists of a single value or a sequence of access paths:

$$\begin{aligned} AP &::= a, a \in Val \\ &::= [](AP])^* \end{aligned}$$

Each access path from the sequence represents the expression for accessing the level of an associative array given by the position of the access path in the sequence. This makes it possible to perform accesses to any associative array of an arbitrary depth where at each level of the array the set of values used for indexing is specified with an arbitrary read-access. As we will see later, a read access using an access path returns a set of values, which can include the *undefined* and the $*$ values.

#	Expression			
(13)	$Eval(x,AP)$	$= \{a\}$	if $AP = a, a \in Val$	
		$= \{values(x,v), v \in Vars(x,AP)\}$	if $AP = [][AP]^*$	
(14)	$Vars(x,AP)$	$= Vars(x, r(x), AP)$		
(15)	$Vars(x, v \in Var, AP)$	$= \{undefVar\}$	if $AP = a, a \in Val$	(a)
		$= \{undefVar\}$	if $AP = [][AP_1][AP_2]...[AP_n] \wedge Vars_n(x,v,AP) = \emptyset$	(b)
		$= Vars_0(x,v,AP)$	if $AP = []$	(c)
		$= Vars_n(x,v,AP)$	if $AP = [][AP_1][AP_2]...[AP_n] \wedge Vars_n(x,v,AP) \neq \emptyset$	(d)
(16)	$Vars_0(x,v,AP)$	$= \{v\}$		
(17)	$\forall_{i \in 1,...,n}$:			
	$Vars_i(x,v,AP)$	$= indices(x, Vars_{i-1}(x,v,AP))$	if $* \in Eval(x,AP_i)$	(a)
		$= indices_r(x, Vars_{i-1}(x,v,AP), Eval(x,AP_i))$	if $* \notin Eval(x,AP_i)$	(b)
(18)	$indices_r(x, V \in \mathscr{P}(Var), I \in \mathscr{P}(Val)) = indices(x,V,I) \cup$			(a)
	$\bigcup_{v \in V, ind \in I} \{u, ((u,\bullet),v) \in indexOf(x) \wedge \nexists_{w \in Vars}((w,ind),v) \in indexOf(x)\}$			(b)

Table 4: Definition of read accesses.

An access path describes an access from a given index (variable). In the following, $[v]([AP])^*$ denotes an access path $[]([AP])^*$ from variable v. Access paths express any PHP expression describing data-access without loss of information. For example, consider the following PHP expressions and the corresponding access paths in the state s: \$$a$[\$$b$]–$[r(s)][a][[r(s)][b]]$, \$\$$a$–$[r(s)][[r(s)][a]]$, \$$a$[\$$b$[\$$c$]][2]–$[r(s)][a][[r(s)][b][[r(s)][c]]][2]$.

3.4 Read Accesses

Tab. 4 defines the read access. *Eval* defines the set of values accessible via an access path from a symbol-table variable at a given state (13). *Vars* defines the set of variables accessible via an access path from the symbol-table variable (14) or from an arbitrary variable (15).

If the access does not identify any variable, the set consists of the undefined variable *undefVar* (15-a)–(15-b). Otherwise the set of variables is obtained by a traversal of the *indexOf* relation. The traversal starts from the specified variable (16). At each level, the access can be performed either with a statically-unknown value (17-a) or with a set of statically-known values (17-b). In the first case, the set of variables at the next level involves all indices of all variables at the current level. Note that these indices include *unknown* fields. If the access is performed with a set of statically-known values, the set of variables at the next level includes indices of all variables at the current level that are identified by the values (18-a). Moreover, if the accessed index is not yet defined for a variable at the current level, the unknown field of the variable is added (18-b). Note that it is not necessary to follow aliases. The reason is that a write access copies the data to all possible targets, including all possible aliases.

If a set of values at each level of an access path contains exactly one value, the *Var* function yields a single variable. We use the notation $[var][v_1]..[v_n]$ in state x to denote the variable $Vars(x, var, [][v_1]...[v_n])$. If the state and the variable from which the access is performed is clear from the context, we write only $[][v_1]...[v_n]$.

Example 4: *Assume the read-access at line 5 in Fig 1. The access is performed from the root variable of the state using the access path* $[][arr][1]$. *The variable* $[][arr]$ *is defined at line 4, while the index* $[][arr][1]$ *of this variable is not defined. Thus, the first level consists of variable* $[][arr]$, *while the second level consists of its unknown field—*$[][arr][\bullet]$.

3.5 Write Accesses

Tab. 6 defines the GEN' set, which contains variables that are created by the assignment and alias statements—the variables statically mentioned for the first time in the left-hand side of the statement[2]. It also defines the variables that are updated by these statements. Tab. 7 defines *KILL* and *GEN* sets, which contains data that are removed and added to these variables. Tab. 5 defines the deep copy of a variable, which is used when a new variable is created and when an existing variable is assigned to another one.

Collecting Variables

Tab. 6 defines four sets of variables. *Must* and *may* (22) are variables that either must or may, respectively, be updated by the assignment statement, $must'$ and may' (23) are variables for the alias statement. If a variable must be updated, a strong update is performed—new information replaces current information. If a variable only may be updated, a weak update is performed—new information is added to the information already present at the variable.

Example 5: *An example of a weak update is the update at line 4 in Fig. 1—it is not statically known which index of the variable* $[][arr]$ *is updated. Weak updates are also performed, e.g., at line 19 in Fig. 1. While variable* $[][arr2][3]$ *is strongly-updated, variables* $[][arr][3]$ *and* $[][alias]$ *that may be aliases of* $[][arr2][3]$ *are only weakly-updated.*

#	Expression
(19)	$deepcopy_{assign}(S \in \Sigma \times Var, T \in \Sigma \times Var)$:
	$(x_s, v_s) \in S \wedge$
	$(x_t, v_t) \in T \wedge$
	$values(x_t, v_t) \supseteq values(x_s, v_s) \wedge$ (a)
	$\forall_{v_{si} \in Var} \forall_{i_{si} \in Val}((v_{si}, i_{si}), v_s) \in indexOf(x_s) \implies ($ (b)
	$v_{ti} = createindex(x_t, v_t, i_{si}) \wedge$
	$deepcopy((x_s, v_{si}), (x_t, v_{ti})) \wedge$
	$(\nexists_{v_{unkn} \in Var}((v_{unkn}, \bullet), v_s) \in indexOf(x) \wedge$
	$v'_{unkn} = createindex(x_t, v_t, \bullet) \wedge$
	$values(x_t, v'_{unkn}) \supseteq \{undefined\}))$
(20)	$deepcopy(S \in \Sigma \times Var, T \in \Sigma \times Var)$:
	$deepcopy_{assign}(S, T) \wedge$
	$(x_s, v_s) \in S \wedge$
	$(x_t, v_t) \in T \wedge$
	$aliases_{must}(x_t, v_t) \supseteq \{a, a \in aliases_{must}(x_s, v_s)\} \wedge$
	$aliases_{may}(x_t, v_t) \supseteq \{a, a \in aliases_{may}(x_s, v_s)\} \wedge$
(21)	$createindex(x, parent \in Var, ind \in Val) = \{var, var = newvar(x) \wedge$
	$indexOf(x) \supseteq \{((var, ind), parent)\} \wedge$
	$values(x, var) \supseteq \{undefined\} \wedge$
	$aliases_{must}(x, var) \supseteq \{var\}\}$

Table 5: Definition of deep copy of the index.

Similarly to read accesses, the variables which are updated by a statement are defined by a traversal of the *indexOf* relation starting in the root variable of the state. The traversal uses the access path of the left-hand side (`LHSAP`). However, the traversal differs from that of a read access. The first difference is

[2]In PHP, variables are defined also when they are mentioned for the first time in the right-hand side of the alias statement, i.e., the statement `$a = &$b` defines the variable `$b` if it is not defined. While we model this behavior in our implementation, we ommit it from the formalization to make the presentation of our approach more clear.

#	Expression
	Assignemnt / Alias: $LHSAP = RHSAP / LHSAP = \&RHSAP$ $LHSAP \sim [][AP_1][AP_2]...[AP_n]$ $\forall_{j=1,2,...,n} I_j = Eval(AP_j)$
(22)	$must = must_n \wedge may = may_n$
(23)	$must' = must'_n \wedge may' = may'_n$
(24)	$indices_w(V \in \mathscr{P}(Var), I \in \mathscr{P}(Val))$ $= indices(IN, V, I) \cup$ $\bigcup_{v \in V, ind \in I} \{defindex(v, ind),\ \nexists_{w \in Var}((w, ind), v) \notin indexOf(IN \cup GEN')\}$
(25)	$defindex(v \in Var, ind \in Val)$ $= \{i, ((u, \bullet), v) \in indexOf(IN \cup GEN') \wedge$ $i = createindex(GEN', v, ind) \wedge$ $deepcopy((IN \cup GEN', u), (GEN', i))\}$
(26)	$must_0 = \{r(IN)\} \wedge may_0 = \emptyset$
(27)	$\forall_{j \in 1,2,...,n}$ $((\lvert I_j \rvert = 1 \wedge I_j \neq \{*\}) \wedge ($ (a) $must_j = aliases_{must}(indices_w(must_{j-1}, I_j)) \wedge$ $may_j = aliases(IN'_k, indices_w(may_{j-1}, I_j)) \cup aliases_{may}(IN, indices_w(must_{j-1}, I_j))))) \vee$ $((\lvert I_j \rvert > 1 \wedge * \notin I_j) \wedge ($ (b) $must_j = \emptyset \wedge$ $may_j = aliases(IN'_k, indices_w(may_{j-1} \cup must_{j-1}, I_j)))) \vee$ $(* \in I_j \wedge ($ (c) $must_j = \emptyset \wedge$ $may_j = aliases(IN'_k, indices(IN'_k, may_{j-1} \cup must_{j-1})))$
(28)	$((I_n \rvert = 1 \wedge I_n \neq \{*\}) \wedge ($ $must'_n = indices^j_w(must_{j-1})) \wedge$ $may'_n = indices^j_w(may_{j-1}))) \vee$ $((\lvert I_j \rvert > 1 \wedge * \notin I_j) \wedge ($ $must'_n = \emptyset \wedge$ $may'_n = indices^j_w(may_{j-1} \cup must_{j-1}))) \vee$ $(* \in I_n \wedge ($ $must'_n = \emptyset \wedge$ $may'_n = indices(IN'_k, may_{n-1} \cup must_{n-1})))$

Table 6: Definition of collecting variables for an update.

that for a write access also the corresponding aliases are followed. That is, all aliases whose data can be possibly changed are updated by the write access. That is why it is not necessary to follow the aliases during read accesses. The second difference is that if a write access to an index identified by a statically known value is performed and this index does not exist, it is created.

Creating new indices when traversing the *indexOf* relation is handled by the definition of the $indices_w$ set (24). Note that write accesses to *unknown* fields in preceding program points could update also the newly defined index and its sub-indices. Thus the new index contains a deep copy of the `unkwnown` field (19). That is, it contains all values and aliases from the *unknown* field (20)–(21-a) and also a deep copy of all indices of the *unknown* field (19-b).

Example 6: *The statement at line 12 in Fig. 1 creates a new variable* $[][arr][1]$. *The write access to the unknown field* $[][arr][\bullet]$ *at line 11 could update also this new variable and the data from this unknown*

field is thus copied to a new variable. Thus the sub-index $[][arr][1][2]$ *of the variable* $[][arr][1]$ *is defined.*

The traversal begins with the $must_0$ set initialized with the variable $r(IN_k)$, which corresponds to the symbol table and the may_0 set initialized with the empty set (26). The statement (27) describes a single step of the traversal at the level `j`. The set I_j of values used to access the j-th level of an associative array can have (27-a) a single statically-known value, (27-b) several statically-known values, or (27-c) it can contain a statically-unknown value.

Example 7: *As an example of case (27-a), see the statement at line 9 in Fig. 1. The* $must_2$ *set consists of variable* $[r(GEN')][arr][2]$*, the* may_2 *set of variable* $[r(IN)][alias]$*, and the set of values* I_3 *consists of value* 3*. Thus, the* $must_3$ *set contains must aliases of the index* $[r(GEN')][arr][2][3]$*. The only must-alias of this index is the index itself. The* may_3 *set contains all aliases of index* $[r(GEN')][alias][3]$*, which is again only the index itself and all may-aliases of index* $[r(GEN')][arr][2][3]$*, which is the empty set.*

In (27-b) and (27-c), the $must_j$ set is empty—it is not known which index is accessed. The may_j set consists of all aliases of the indices of variables in $must_{j-1}$ and may_{j-1}. In case of (27-c), the variables do not need to be identified by values of I_j and no new variables are created.

Example 8: *As an example, see the statement at line 12 in Fig. 1. At the second level,* $must_2$ *set consists of the variable* $[r(GEN')][arr][1]$ *and the* may_2 *set is empty. The* $must_3$ *set is empty, while the* may_3 *set consists of variables* $[r(GEN')][arr][1][2]$ *and* $[r(GEN')][arr][1][\bullet]$.

The difference between computing variables that will be updated by the assignment and the alias statement (28) is caused by the fact that the assignment statement updates the variable and all its aliases with new values while the alias statement un-aliases the variable from all its original aliases while keeping their values unaffected. However, the alias statement respects the aliases at previous levels the same way as the assignment statement. In other words, the expressions for obtaining indices to be updated for the assignment and the alias statements treat differently only the last level.

Example 9: *In the case of the alias statement at line 15 in Fig. 1, both variables* $[r(GEN')][arr][2][1]$ *and* $[r(GEN')][alias][1]$ *will be updated, since (*$[r(GEN')][arr][2]$ *is a may-alias of* $[r(GEN')][alias]$*). In the case of the alias statement at line 16, only the variable corresponding to* $[r(IN)][arr][2]$ *will be updated—the variable* $r(IN)][arr]$ *has no alias.*

Performing Update

Tab. 7 defines how the collected variables are updated by the assignment and alias statements. Expressions (29)–(31) describe the data that is removed from the variables. Both the alias and assignment statements remove all the values and indices of all updated variables that were present in these variables before the update including the data added in collecting phase (29)–(30). The alias statement also removes aliasing data (31).

Expressions (32)–(35) describe the data that is added to the variables. In the case of a strong update (32), both the statements add just the data that results from merging variables obtained by the read accesses of the right-hand-side access path (RHSAP). In the case of a weak update, the original variable is merged too, so the original data is preserved (33). Technically, the update is described as first merging the data to a temporary fresh variable and then copying it from this variable to the variable being updated (34)–(35). The difference between the assignment and the alias statements is that while the former one does not copy the alias data at the first level (34), the latter one does (35). Note that for the other levels, the alias data are copied also in the case of the assignment statement (20-b).

Example 10: *As an example, see the update at line 20 in Fig 1. The must set is empty, the may set consists of variables* $[][arr][1]$, $[][arr][2]$, $[][arr][\bullet]$, $[][arr2][1]$, $[][arr2][2]$, $[][arr2][3]$*, and* $[][arr2][\bullet]$*. Consider the update of variable* $[][arr][1]$*. Because the update is weak, new data results from merging the result of read access of RHSAP, which is* $[][arr2]$*, with the variable that is updated, which is* $[][arr][1]$.

#	Expression	
	Assignemnt / Alias:	$LHSAP = RHSAP / LHSAP = \&RHSAP$
(29)	$\forall_{v \in must \cup may \cup must' \cup may'}$	$values(KILL, v) = values(IN'_k, v)$
(30)	$\forall_{v \in must \cup may \cup must' \cup may'}$	$indexOf(KILL, v) = indexOf(IN'_k, v)$
(31)	$\forall_{v \in must'}$	$aliases(KILL, v) = aliases(IN'_k, v)$
(32)	$\forall_{v_t \in must \cup must'}$	$src = \{(IN, v), v \in Vars(IN, RHSAP)\}$
(33)	$\forall_{v_t \in may \cup may'}$	$src = \{(IN, v), v \in Vars(IN, RHSAP)\} \cup \{(IN', v_t)\}$
(34)	$\forall_{v_t \in must \cup may}$	$deepcopy_{assign}((GEN', v = fresh(GEN')), (GEN, v_t)) \wedge mergeVars((GEN', v), src)$
(35)	$\forall_{v_t \in must' \cup may'}$	$deepcopy((GEN', v = fresh(GEN')), (GEN, v_t)) \wedge mergeVars((GEN', v), src)$

Table 7: Definition of updates for assignment and alias statement and new object expression.

Consequently, after the update, the variable $[]arr[1]$ *contains indices* $[][arr][1][1]$, $[][arr][1][2]$, *and* $[][arr][1][3]$. *E.g., index* $[][arr][1][1]$ *contains the data merged from indices* $[][arr]\ [1][1]$ *and* $[][arr2][1]$.

Example 11: *Now consider the update at line 17. The must set consists of* $[]\ [arr2]$, *the may set is empty. The read-access of the RHSAP results in reading* $[][arr]$. *Because the update is strong,* $[][arr]$ *is the only variable that is merged and thus it is only deep-copied. Note that in the case of the assignment statement, the alias data is copied for all the levels except for the first one. Thus, because of the alias statement at line 16,* $[][arr2][2]$ *is a must-alias of* $[][alias3]$ *and* $[][arr][2]$ *and due to the alias statement at line 4, e.g.,* $[][arr2][\bullet]$ *is a may-alias of* $[][alias]$ *and* $[][arr2][\bullet]$. *Consequently, the statement at line 18 strongly updates not only* $[][arr2][2]$, *but also* $[][arr][2]$ *and* $[][alias3]$. *Similarly, the statement at line 19 strongly updates* $[][arr2][3]$ *and weakly updates* $[][arr][\bullet]$ *and* $[][alias]$. *Thus, the subsequent read access using access path* $[][arr][3]$ *would read also value* 9.

3.6 Merge

Tab. 8 defines the merge operation. Expression (36) defines the operation *mergeStates*, which is used in the first data-flow equation to define the *IN* state of a node. It merges the root variables of the *OUT* states of all predecessors of the node to the root variable in the IN state. Note that if the node has only one predecessor, the merge actually corresponds to a deep copy.

Expression (37) defines how variables in given states are merged into the resulting state. Note that this operation is used also when an update is performed (34)–(35). In (37-a), for each variable being merged and a state in which it is defined, the access paths of all sub-indices of the variable are collected. The empty access path [], which corresponds to the variables being merged, is added to these access paths (37-a).

Example 12: *For merging at the join point at line 13 in Fig. 1 and for the symbol-table variable of the first branch, the following access paths are collected:* $[][alias]$, $[][alias][2]$, $[][alias][3]$, $[][arr]$, $[][arr][\bullet]$, $[][t]$, $[][t][2]$, $[][arr][1]$, $[][arr][1][2]$, $[][arr][1][3]$, $[][arr][2]$, $[][arr][2][3]$.

Sub-expression (37-b) further extends the set of access paths. After the extension, it contains an access path for each variable that will be defined in the resulting state. For each access path that contains the value $\bullet$ (corresponding to the *unknown* field) at a certain level it adds the access paths that are created from this access path by replacing the value $\bullet$ with all the values that are in the input access paths at this level. Note that this adds new access paths to the resulting set only if there were performed corresponding statically-known write accesses from different variables being merged. This is analogous to copying indices of the *unknown* field when there is a write access with a given value for the first time and a new variable is thus defined. While write-accesses to *unknown* fields in preceding program points could also create sub-indices of a newly defined variable, in the case of merge there could be write-accesses to *unknown* fields that could create sub-indices of variables created elsewhere. Both these

#	Expression
(36)	$mergeStates(OUT \in \mathscr{P}(\Sigma)) = \{IN, mergeVars((IN, r(IN)), \{(o, r(o)), o \in OUT\})\}$
(37)	$mergeVars(R \in \Sigma \times Var, M \in \mathscr{P}(\Sigma \times Var))$:
	$APs = \bigcup_{(x,v_r)\in M}(accessPaths(x, v_r, [])) \cup \{[]\} \wedge$ (a)
	$ResAPs = extend(APs) \wedge$ (b)
	$\forall_{AP \in ResAPs}(mergeAP(R, M, AP))$ (c)
(38)	$mergeAP(R = (x_R \in \Sigma, v_R \in Var), M \in \mathscr{P}(\Sigma \times Var), AP)$:
	$resVar = createVar(R, AP) \wedge$ (a)
	$mergedVars = \bigcup_{(x_M,v_M)\in M}\{(x_M, Var(x_M, v_M, AP))\} \wedge$ (b)
	$values(o, resVar) = \bigcup_{(x_m,v_m)\in mergedVars}\{values_{undef}(x_m, v_m)\} \wedge$ (c)
	$aliases_{must}(x_R, resVar) = \bigcap_{(x_m,v_m)\in mergedVars}\{aliases_{must}(x_m, v_m)\} \wedge$ (d)
	$aliases_{may}(x_R, resVar) = \bigcup_{(x_m,v_m)\in mergedVars}\{aliases_{must}(x_m, v_m) \cup aliases_{may}(x_m, v_m)\} -$ (e)
	$aliases_{must}(x_R, resVar) \wedge$
	$indices(x_R, resVar) = \bigcup_{(x_m,v_m)\in mergedVars}\{createVar((x_R, resVar), [][n]),$ (f)
	$n \in indicesNames(x_m, v_m)\}$
(39)	$accessPaths(x \in \Sigma, v_R \in Var, AP) = \bigcup_{((i,i_{name}),v_R)\in indexOf(x)}($
	$\{AP[i_{name}]\} \cup accessPaths(x, i, AP[i_{name}]))$
(40)	$extend(APs) = APs \cup \bigcup_{l\in levels(APs)\wedge(AP\in APs\wedge level(AP)>l)}\{newAP(AP, l, values(APs, l))\}$
(41)	$newAP([][v_1]...[v_n], l \in Int, V \in \mathscr{P}(Val)) = \{[][u_1]...[u_n], \forall_{i=1,...,n\wedge i\neq l}(u_i = v_i) \wedge u_l \in V\}$ if $v_l = \bullet$
	$= \emptyset$ if $v_l \neq \bullet$
(42)	$level([][v_1]..[v_n]) = n$
(43)	$levels(APs) = \{l, l = level(AP) \wedge AP \in APs\}$
(44)	$value([][v_1]..[v_n], l \in Int) = v_l$
(45)	$values(APs) = \{v, v = value(AP) \wedge AP \in APs\}$
(46)	$indicesNames(x, var) = \{n, \exists_{iv\in Var}\exists_{n\in Val}((iv, n), var) \in indexOf(x)\}$

Table 8: Definition of merge.

operations are thus necessary to preserve the invariant that if there could be a write-access to an index using a statically known value at a given level, all the data that could be possibly written to this index are stored there.

Example 13: *When the merge at line 13 in Fig. 1 is performed, the set of access paths is extended with* $[][arr][2][2]$*, which then causes the corresponding variable in the resulting state to be created. The reason is that while in the else branch* $[][arr][2]$ *is not defined, there is a write access to the unknown field at line 11 that could create a sub-index of this variable. The read access using* $[][arr][2][2]$ *will follow* $[][arr][\bullet]$ *in the second level and will finally* $[][arr][\bullet][2]$*, which contains values* 6 *and undefined. In the then branch,* $[][arr][2]$ *is created and it will be thus added to resulting state of the merge. The read access follows this variable at the second level. Thus, to access value* 6 *with access path* $[][arr][2][2]$*, there must be a variable* $[][arr][2][2]$ *which contains this value in the resulting state. This is analogous to copying data from the unknown field to a variable that is statically stated for the first time when the update is performed, e.g, a new variable* $[][arr][1][2]$ *containing values* 6 *and undefined is created during the update at line 12.*

Sub-expression (37-c) merges variables corresponding to an access path in the merged states to the variable which corresponds to this access path in the resulting state. First the variable in the resulting state using the access path and the output variable is created (38-a). Then, the corresponding variables in merged states are obtained (38-b). Finally, the data of these variables are merged to the resulting variable (38-c)–(38-f). Note that while in the case of (38-a), the access path is used to create the variable which

directly corresponds to the access path, in the case of (38-b) the access path is used to get variables in merged states by the read access (15). That is, in the case of (38-b), the variables can be accessible by *unknown* fields even at levels where the access path contains a static value. Note that both (38-a) and (38-f) contain the expression *createVar*, however, resulting variables are created only once—if expression *createVar* is used the second time with the same arguments, it returns the existing variable.

Example 14: *As an example, see the merge corresponding to the join point at line 13 in Fig. 1. For the access path* $[][arr][1][3]$*, variable* $[][arr][1][3]$ *is be created in the resulting state. The merged variable in the first branch is* $[][arr][1][3]$*, however, for the second branch, the data corresponding to this access path is located in* $[][arr][1][\bullet]$*. For the access path* $[][arr][2][2]$*, a variable is created in the resulting state, the merged variables are undefVar in the first branch and* $[][arr][\bullet][2]$ *in the second branch.*

3.7 Termination and Soundness

Termination: The values in our model are represented either by constants present in the program or values $*$, *undefined*, and $\bullet$. Thus the number of values is finite. From this it follows that the number of defined indices in a single dimension of arrays is finite. However, due to the presence of loops and recursion, the infinite number of dimension may be generated. To ensure the termination, the number of dimensions must be limited. This can be done either explicitly [5] or implicitly by using, e.g, allocation-site abstraction for creating new dimensions of arrays. Then, the total number of indices is finite and *alias* and *indexOf* relations are finite as well. The transfer functions defined in Tab. 6, Tab. 7, and Tab. 8 are monotonic so the fixpoint computation terminates.

Our approach is implemented as a part of a static analyzer with the support of operators such as $+$. Thus, potentially an infinite number of values can be generated in the program due to presence of loops and recursion. To ensure termination of fixpoint computation in this case, it is necessary to limit the size of value sets of each variable by a constant—larger value sets would be represented either by value $*$ or by a finite abstract domain.

Soundness: We use the following soundness argument:

> If a value can be written to a given variable (index) by a write-access at the node n_1 of CFG, it is read from this variable by a read access in node n_2 of CFG that follows n_1 if and only if there is a path from n_1 to n_2 in CFG where the variable is not strongly-updated by different value. Moreover, if there is a path from the initial node to a given node such that a variable was not strongly-updated, the set of the variable values returned by a read access always includes the *undefined* value.

Note that a value can be written to a given index also if the write access is statically unknown at any level. Also note that there can be an arbitrary number of join points between n_1 and n_2. We do not provide the proof of the argument, however, its validity follows from definitions of read accesses, write accesses, and merge.

4 Implementation and Evaluation

We implemented our approach in the context of the Weverca [2] static analyzer. Besides associative arrays, the implementation supports also objects. Weverca makes it possible to perform static taint analysis [11] as well as other analyses that can be used for finding bugs and evaluation the system safety. It constructs CFGs on the fly. If the processed program point is a virtual method call, it uses a read-access to get the method's receiver object and uses its type to determine the method to be called. It also uses a read-access to get values of variables that specify the name of a method and the include target in case of a dynamic method call or dynamic include. Weverca also implements several options of context-sensitivity for function and method calls.

n	CFG nodes (mCODE_n/CODE_n)	Variables	Analysis Time (s) (mCODE_n/CODE_n)
1	235 / 117	107	0.4 / 0.3
2	463 / 231	211	1.3 / 0.7
3	919 / 459	419	4.9 / 2.5
4	1831 / 915	835	22.8 / 10.3

Table 9: The evaluation results.

The novelty of our approach is that it is sound and precise even if statically-unknown data from the input are used to access associative arrays-like structures at an arbitrary level. Other approaches, such as [10] are more limited, e.g., they model only associative arrays of the depth 1. Since it is not an issue to implement a precise analysis but a precise scalable analysis, a question we want to answer is whether our approach scales well. In particular, we want to know how the merge function scales with respect to the number of variables being merged in the presence of dynamic write accesses to multiple levels of associative arrays.

To evaluate our approach we used the code `CODE_n` that was generated from the code fragment at lines (2)–(19) in Fig. 1 replicated 2^n times with all the variables except for the variable `$any` with the prefix unique for each replica. This code contains non-trivial dynamic accesses to multiple levels of associative arrays; the number of variables in this code grows exponentially with n. However, the number of variables that are being merged at each join point is constant. To evaluate the complexity of the merge operation, we use the code `mCODE_n` (Fig 2) which is defined using `CODE_n`. In `mCODE_n`, all variables defined in the code are involved in the top-level merge operation.

```
$any = $_GET['user_input'];
if ($any) { CODE_n }
else { CODE_n }
```

Figure 2: The code used for the evaluation

Tab. 9 shows the results of analysis of `mCODE_n` and `CODE_n` for different values of n. The table shows the number of nodes of generated CFG, the number of variables defined in our representation in the OUT state of the program end point, the running time of the analysis. The comparison of running times of `mCODE_n` and `CODE_n` shows that the merge operation is efficient even if all the variables all involved—`mCODE_n` analyzes two copies of `CODE_n` and additionally merges all the variables. Thus the increase of time consumption is caused mostly by the increase of the number of variables. This is caused mainly by the fact that the amount of data stored in a state grows with the number of variables defined in the state and that in the implementation there is no sharing of data between the states. We believe that by optimizing the implementation, the scalability can be highly improved and the approach can scale up to thousands of variables and tens of thousands nodes of CFG.

5 Related work

In this section, we discuss tools and techniques related to the area of our interest.

Pixy [4] 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 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 differences from programming languages such as Java and C. Alias analysis introduced in Pixy incorrectly models aliasing between arrays and array indices. Web applications use associative arrays and objects extensively, thus we believe that this is an

essential limitation. Importantly, Pixy does not perform type inference, which also limits its precision and soundness.

Stranger [14] 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.

Jang [3] presents flow-insensitive points-to analysis for JavaScript. Their work is closely related to our approach, since JavaScript provides dynamic features similar to those in PHP. The same way as our approach, they model variables, arrays, and objects using associative arrays. However, they precisely model only assignments to constant indices and they use special unknown field for all other assignments. Moreover, the same as Sridharan [10], they limit the accesses to associative arrays to depth one. Next, PHP supports creating aliases between variables which JavaScript does not. Therefore, we must maintain alias information to perform updates of points-to information correctly.

Phantm [5] is a PHP 5 static analyzer for type mismatch based on data-flow analysis; it aims 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, 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.

F4F [9] focuses on static taint analysis of web applications that use frameworks. This work uses a semi-automatically generated specification of framework-related behaviors to reduce the amount of statically-unknown information, which arises, e.g., from reflective calls.

Andromeda static taint analyzer [12] fights the problem of scalability of taint analysis by computing data-flow propagations on demand. It uses forward data-analysis to propagate tainted data and ignores propagation of other data. If tainted data are propagated to the heap, it uses backward analysis to compute all targets to which the data should be propagated. Andromeda analyzes Java, .NET, and JavaScript applications. The drawback of the approach is that it propagates only taint information. The control-flow of the application can depend on other information which are then not available.

Sridharan [10] et. al. present static flow-insensitive points-to analysis for JavaScript. They model objects in JavaScript using associative arrays that can be accessed by arbitrary expressions. However, they limit the accesses to depth one. They show that in this setting, the complexity of flow-insensitive points-to analysis becomes $O(N^4)$, where N is the program size, in contrast to the $O(N^3)$, which is the case when the accesses are constant. To enhance the precision and scalability of the analysis, they identify correlations between dynamic property read and write accesses. If the updated location and stored value can be accessed by the same first class entity (variable), it is extracted to a function parametrized by this entity; this function is then analyzed context-sensitively with the context be the variable. Thus, the correlation between the update and store is preserved.

Schafer [8] et. al. present a dynamic analysis for identifying variables and expressions that always have the same value at a given program point and finding this value. Such values can be used, e.g, to make a constrained dynamic language constructs static and thus enhance the scalability and precision of static analysis.

Wei [13] et. al. reduce the number of statically-unknown information in static analysis by collecting such information at run-time.

Livshits [6] et. al. propose a method of fully automatic placement of security sanitizers and declassifiers. They place sanitizers statically whenever possible and they try to minimize the amount of run-time tracking. The input of their analysis is a data-flow graph generated by a static analyzer. The quality of sanitization placement—the reduction of the amount of run-time tracking—depends on the quality of

this data-flow graph.

6 Conclusion and future work

Dynamic languages such as PHP contain features that pose significant challenges for static analysis. In our previous position paper [1] we introduced our approach to static analysis of PHP and described particular parts of the Weverca analyzer [2]. In this paper we focused on the data modeling part. We described dynamic accesses to associative array-like structures, which make it hard to apply value and points-to analysis, and presented our approach to this challenge. We focused on soundness and precision—we do not want to overwhelm the user with too many false alarms, which is often the case of related tools.

The evaluation shows that the prototype implementation of our approach scales to hundreds of variables and thousands of nodes in the control-flow graph. We believe that the scalability of the tool can be further enhanced. Therefore, as future work, we plan to enhance the scalability by implementing optimizations, in particular, sharing the data between nodes of the control-flow graph, and then to evaluate the scalability on real-life applications.

References

[1] David Hauzar & Jan Kofroň (2012): *On Security Analysis of PHP Web Applications*. In: *STPSA 2012*, IEEE, pp. 577–582, doi:10.1109/COMPSACW.2012.106.

[2] David Hauzar & Jan Kofroň (2014): WEVERCA. `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`.

[3] Dongseok Jang & Kwang-Moo Choe (2009): *Points-to analysis for JavaScript*. SAC '09, ACM, New York, NY, USA, pp. 1930–1937, doi:10.1145/1529282.1529711.

[4] N. Jovanovic, C. Kruegel & E. Kirda (2006): *Pixy: a static analysis tool for detecting Web application vulnerabilities*. In: *S&P'06*, IEEE, doi:10.1109/SP.2006.29.

[5] Etienne Kneuss, Philippe Suter & Viktor Kuncak (2010): *Runtime Instrumentation for Precise Flow-Sensitive Type Analysis*. In: *RV*, pp. 300–314, doi:10.1007/978-3-642-16612-9_23.

[6] Benjamin Livshits & Stephen Chong (2013): *Towards Fully Automatic Placement of Security Sanitizers and Declassifiers*. POPL '13, ACM, New York, NY, USA, pp. 385–398, doi:10.1145/2429069.2429115.

[7] Flemming Nielson, Hanne R. Nielson & Chris Hankin (1999): *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, doi:10.1007/978-3-662-03811-6.

[8] Max Schäfer, Manu Sridharan, Julian Dolby & Frank Tip (2013): *Dynamic Determinacy Analysis*. PLDI '13, ACM, New York, NY, USA, pp. 165–174, doi:10.1145/2499370.2462168.

[9] Manu Sridharan et al. (2011): *F4F: Taint Analysis of Framework-based Web Applications*. OOPSLA '11, ACM, New York, NY, USA, pp. 1053–1068, doi:10.1145/2048066.2048145.

[10] Manu Sridharan et al. (2012): *Correlation Tracking for Points-to Analysis of Javascript*. ECOOP'12, Springer-Verlag, Berlin, Heidelberg, pp. 435–458, doi:10.1007/978-3-642-31057-7_20.

[11] Omer Tripp et al. (2009): *TAJ: Effective Taint Analysis of Web Applications*. PLDI '09, ACM, New York, NY, USA, pp. 87–97, doi:10.1145/1542476.1542486.

[12] Omer Tripp et al. (2013): *ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications*. FASE'13, Springer-Verlag, Berlin, Heidelberg, pp. 210–225, doi:10.1007/978-3-642-37057-1_15.

[13] Shiyi Wei & Barbara G. Ryder (2013): *Practical Blended Taint Analysis for JavaScript*. ISSTA 2013, ACM, New York, NY, USA, pp. 336–346, doi:10.1145/2483760.2483788.

[14] Fang Yu, Muath Alkhalaf & Tevfik Bultan (2010): *Stranger: An automata-based string analysis tool for PHP*. *TACAS'10*, doi:10.1007/978-3-642-12002-2_13.