Framework for Static Analysis of PHP Applications*

David Hauzar and Jan Kofroň

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

Abstract

Dynamic languages, such as PHP and JavaScript, are widespread and heavily used. They provide dynamic features such as dynamic type system, virtual and dynamic method calls, dynamic includes, and built-in dynamic data structures. This makes it hard to create static analyses, e.g., for automatic error discovery. Yet exploiting errors in such programs, especially in web applications, can have significant impacts. In this paper, we present static analysis framework for PHP, automatically resolving features common to dynamic languages and thus reducing the complexity of defining new static analyses. In particular, the framework enables defining value and heap analyses for dynamic languages independently and composing them automatically and soundly. We used the framework to implement static taint analysis for finding security vulnerabilities. The analysis has revealed previously unknown security problems in real application. Comparing to existing state-of-the-art analysis tools for PHP, it has found more real problems with a lower false-positive rate.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Static analysis, abstract interpretation, dynamic languages, PHP, security

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.689

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.1.1.11`

1 Introduction

To analyze programs precisely and soundly, static analysis needs to resolve method calls, include statements, and accesses to data structures. Since in dynamic languages, targets of method calls and include statements can depend on information about values (and types) of expressions, value analysis tracking values of all primitive data types present in the language needs to be performed. Moreover, due to frequent use of dynamic data structures such as associative arrays and objects, value analysis needs to be combined with heap analysis. These depend on each other also the other way round – since array indices and object properties can be accessed using arbitrary expressions, heap analysis needs value analysis to evaluate these expressions. This makes any end-user static analysis that takes this into account overly complex.

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, http://www.spark-2014.org/proofinuse) of the French national research organization.

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 689–711

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Propagation of tainted data.

Lattice	$(L, \sqsubseteq, \sqcup)$	$(Bool, \implies, \vee)$	
Initial value	$init(v)$	$true$	if $v \in \$\_\texttt{POST} \cup \$\_\texttt{GET} \cup \ldots$
		$false$	otherwise
Transfer function	$[\![LHS = RHS]\!]$	$var = \bigvee_{r \in RHS} r$	if $var \in LHS$
		$var = var$	otherwise
	$[\![st]\!]$	$var = var$	if st is not assignment

In this paper we present a static analysis framework for languages with dynamic features [10] based on abstract interpretation [1]. The framework automatically resolves dynamic features and makes it possible to define static analyses without taking these features explicitly into account.

In particular, our contributions include:

- The architecture of the static analysis framework for dynamic languages and the way dynamic features are automatically resolved.
- Description of all necessary analyses that are needed to automatically resolve dynamic features. We define value analysis that tracks values of all primitive data types of PHP. We articulate our assumptions on heap analysis to take dynamic index and property accesses into account – indices and properties are created when they are accessed for the first time and accesses can be performed using arbitrary value expressions, yielding statically unknown values.
- Composition of all necessary analyses allowing to define these analyses independently. Here the main challenge is defining the interplay of value analysis and heap analysis taking dynamic features into account. The composition is sound; if the analyses being composed are sound, the resulting analysis is sound as well.

2 Motivation

As a motivation example, consider static taint analysis, which is often used for security analysis of web applications. It can be used for detection of security problems, e.g., vulnerability of an application to SQL injection and cross-site scripting attacks. Static taint analysis can be described as follows. 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 to a browser, 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. Tainted data are *sanitized* if they are processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potentially malicious parts. 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 specification of forward data-flow analysis computing the propagation of tainted data is shown in Tab. 1[1]. The analysis is specified by the lattice of data-flow facts and lattice operators, the initial values of variables, and the transfer function.

Consider now the code in Fig. 1. The code contains two vulnerabilities to XSS attack [7]. The first vulnerability corresponds to the call at line (25), the second vulnerability corresponds

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

```
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']);
```

Figure 1 Running example.

to the call at line (26). In both cases the method `show` of `Templ1` can be called (line (5)) with the parameter `$msg` being tainted and going to the sink. Taint analysis defined using our framework uses just the information in Tab. 1 and can still detect both vulnerabilities. This is possible only because the framework automatically resolves control flow and accesses to built-in data structures. That is, the framework computes that the variable `$t` can point to objects of types `Templ1` and `Templ2` and that the variable `$mode` can contain values `"show"` and `"log"`. Based on this information, it automatically resolves calls at lines (25) and (26). As the framework automatically reads the data from and updates the data to associative arrays and objects, tainted data written at line (23) are read at line (25). Moreover, at line (24), the tainted data are automatically propagated to index `$users['admin']['addr']` defined at line (11). Consequently, the access of this index at line (26) reads this tainted data.

3 Static Analysis Framework

The architecture of the framework is shown in Fig. 2. 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 properties and evaluates expressions used for accessing data. The control flow is captured in the intermediate representation (IR), while the other information can be accessed using the data representation. In the second phase, end-user analyses of the constructed IR are performed.

Data representation allows accessing analysis states. In particular, it allows reading and writing values, and modifying shape of data structures. Next, it performs join and widening of analysis states and defines their partial order. Importantly, data representation defines

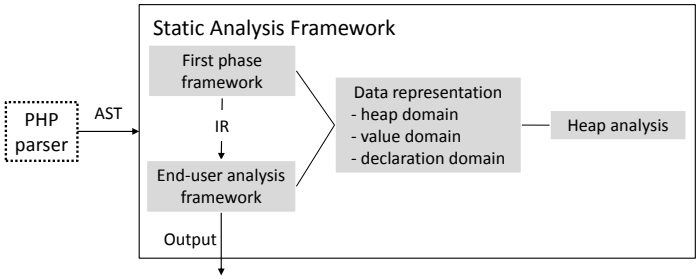

Figure 2 Architecture of the framework.

the interplay of heap, value, and declaration analyses allowing each analysis to define these operations independently on other analyses.

The implementation of heap analysis tracks the shape of the heap and must provide information that the value analysis needs to read values from data structures, update values to data structures, and to join values stored in data structures.

The implementation of the first phase must provide information necessary for computing control flow of the program and the information that the heap analysis needs to access data. That is, it must define value analysis that tracks values of PHP primitive types, evaluates value expressions modeling native operators, native functions, and implicit conversions. Next, the implementation must define declaration analysis handling declarations of functions, classes, and constants. Finally, it must compute targets of throw statements, include statements, and function and method calls.

The implementations of end-user analyses define additional value analyses. In contrast to value analysis for the first phase, which must track values of PHP primitive types, end-user value analyses can use 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 (i.e., value expressions specifying data accesses are evaluated). That is, all information that the data representation needs to determine accessed variables, array indices, and object properties is available.
3. Data representation combines heap, value, and declaration analyses automatically.

3.1 Intermediate Representation

The intermediate representation (IR) of our analysis is a graph, in which each node contains an instruction. There are two types of nodes in the graph – *value nodes* and *non-value nodes*. Value nodes compute and store representation of values while non-value nodes perform other actions. The graph has two types of edges. *Flow edges* represent potential control flow between instructions of the program – they define ordering in which program instructions can be executed. *Value edges* connect nodes that use values (e.g., operators) with nodes that represent these values (e.g., operands).

Each node has associated an analysis state stored in data representation. The state is modified by transfer function defined for the node and the resulting state is propagated to successor nodes connected with flow edges. If a node has more predecessors the states of predecessors are joined.

Note that transfer functions for most of the value nodes are defined as identity – they do not modify the analysis state. That is, most of the value nodes just compute values (e.g., evaluate expressions) or compute information that specify data access to values (e.g.,

compute possible names of variables that they represent). This information is stored in data representation, but it is not a part of the analysis state and thus it is not propagated to successor nodes. Instead, nodes that use these values (e.g. operator nodes) are connected with value nodes (e.g. operands) using value edges. If an operand value is needed when evaluating the operator, the value edge is used to get the value from the operand.

▶ **Example 1.** As an example, consider the intermediate representation corresponding to the statement `$$a=b($c)`. The statement assigns the value computed by function `b` to a variable with the name given by the value of variable `$a`. The resulting intermediate representation is depicted in Fig. 3. Note that the node corresponding to the assignment instruction is connected using a value edge with the source of the assignment (the node containing the value computed by the function `b`) and with the target of the assignment (the node representing the assigned variable – `$`). Next, the latter node is connected using a value edge with the node representing possible names of the assigned variable (node `$a`).

The nodes can be of different types. In the following, we denote value nodes by adding superscript V; for each node, its parameters are the value nodes that are connected with the node using value edges:

variable$^V[n^V]$**:** represents a variable – stores the information necessary for accessing the variable in data representation. The parameter n^V is the value node that represents the name of the variable. Note that reading n^V yields an arbitrary value from the abstract string domain and can thus represent more concrete string values – names. Consequently, the variable node can represent more concrete variables.

property-use$^V[o^V, f^V]$**, index-use**$^V[a^V, i^V]$**:** property-use^V stores the information for accessing a property of the given object. Parameter o^V is the value node storing the representation of the object and f^V is the value node storing the name of the property. Again, reading o^V and f^V yields abstract values and the property-use^V node can get representation of more properties. The index-use^V is similar and it is used for accessing arrays.

assign$^V[l^V, r^V]$**:** represents the assignment of the right operand r^V to the left operand l^V and stores the information for accessing this value. While the parameter l^V is a value node whose type can be variable, property-use, and item-use, the parameter r^V is an arbitrary value node.

alias$^V[l^V, r^V]$**:** represents the alias statement. The alias statement is similar to the assignment statement. However, besides performing the assignment, the alias statement creates explicit alias between its parameters and both parameters of the alias statement must be variables, object properties, or array indices.

expression$^V[e, o_1^V, ..., o_n^V]$**:** represents the expression e with operands $o_1^V, ..., o_n^V$. It stores the representation of the result.

assume$[c]$**:** represents assumption implied, e.g., by *if* and *while* statements. It indicates whether the condition c is satisfiable. If the condition is unsatisfiable, the flow is not propagated to descendant nodes. If the condition is satisfiable, the analysis state is refined according to the condition and then propagated to descendant nodes.

constant-declaration$[d]$**:** represents declaration of a constant.

function-declaration$[d]$**:** represents declaration of a function.

class-declaration$[d]$**:** represents declaration of a class.

call$^V[n^V, o^V, a]$**, construct**$^V[n^V, a]$**:** represents a call of a function whose name is specified using the value node n^V on an object specified using the value node o^V with arguments specified using a list of value nodes a. The construct^V nodes are similar to call^V nodes

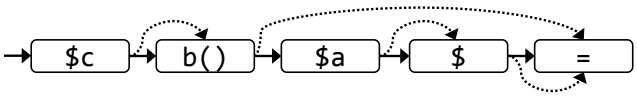

Figure 3 Intermediate representation of the statement `$$a=b($c)`. Solid edges are flow edges, dashed edges are value edges.

and are used for `new` expressions. Note that reading n^V, o^V, and elements of a yields abstract values that can represent more concrete values.

return$[e^V]$**:** represents a return from a function. e^V represents the value of a return expression.

include$[p^V]$**:** represents the inclusion of the script given by the path specified by the value node p^V. Again, a path can represent more concrete values.

eval$[c^V]$**:** evaluates the code fragment specified by value node c^V.

native-method$^a[]$**:** represents execution of a native method or a native function with arguments specified using a list of value nodes a.

extension$[f, a]$**:** is used to dynamically extend the control from IR node f. This is necessary when the information needed to determine the control flow from the node is computed by the analysis. This is the case of calls to functions, methods and constructors, and the include and eval statements. During analysis, for each dynamically discovered control flow from the node, a single extension node is added. Parameter f is the node that is extended. Parameter a is used in the cases that the control flow is extended because of a function, method, and constructor call and it is a list of value nodes representing parameters of the call.

extension-sink$[n]$**:** represents a join point of all the extensions of node n.

try-scope-start$[c]$ **and try-scope-end**$[c]$: represent the start and the end of a `try` block. Parameter c represents catch blocks associated with the `try` block.

throw$[v^V]$**:** represents the `throw` statement. Parameter v^V is the node representing the value to be thrown.

catch$[v^V]$**:** represents a catch block. It contains a node representing the first node of the catch block as a flow child. Parameter v^V is the node representing the value to be thrown.

3.2 Building IR

To determine control flow of the analyzed application, the information from value analysis is needed. Thus, the IR is built gradually during the analysis.

Initially, IR for the entry script of the application (typically `index.php`) is built. This IR contains caller nodes – the nodes corresponding to function, method, and constructor calls, script inclusions, and eval statements. Since at this point, the information needed to compute control flow from these nodes is not yet available, the control flow is initially directed to the nodes that follow the calls.

The control flow is then extended during static analysis. When processing a caller node, the analysis framework provides the first phase implementation with all information already computed by the analysis that is relevant to determine the control flow. Using this information, the first phase implementation finds appropriate function or method definitions or scripts to be included, and it computes IRs representing their control flow. The first phase implementation can build new IRs or use existing IRs, which are then shared among multiple caller nodes. This way, the first phase implementation can control context sensitivity. Finally, the control flow of the caller nodes is extended with computed IRs.

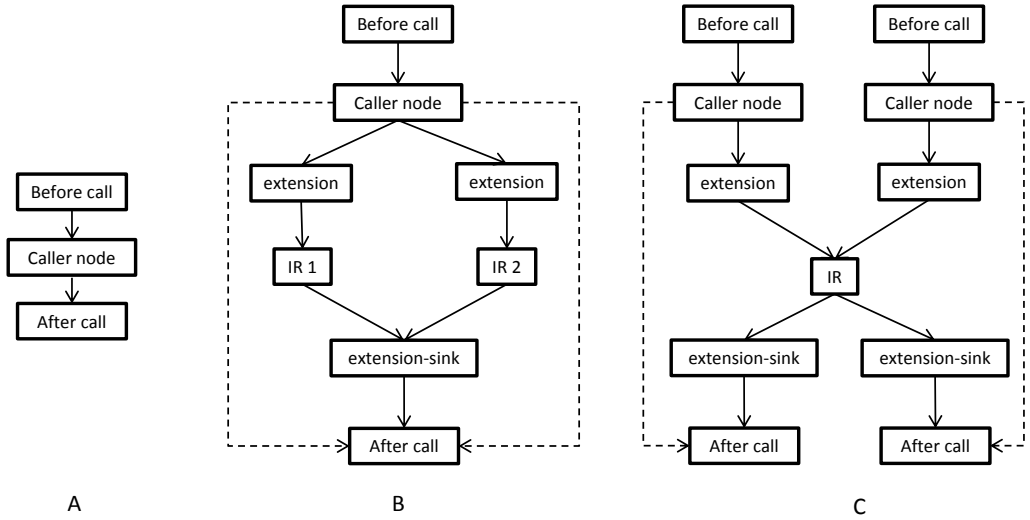

Figure 4 Building IR. Initial IR – the control flow of the caller node has not yet been extended (A). IR after processing the caller node during static analysis. The control flow of the caller node is extended with two IRs – IR 1 and IR 2 (B). Single IR shared between multiple caller nodes (C).

IRs are not connected to caller nodes directly – extension node is inserted between each caller node and the entry node of the connected IR and extension-sink node is inserted between each final node of the IR and the node following the call. While an extension node binds actual parameters to formal parameters for function, method, and constructor calls, an extension-sink joins states of final nodes of all the IRs that extend the corresponding caller node.

▶ **Example 2.** Fig. 4-A shows IR after it is initially built. Fig. 4-B shows IR after extending the caller node during the analysis. In this case, the caller is extended with more IRs – this can happen, e.g., if a method is called on an object that can be of more types. Fig. 4-C shows the case when a single IR is shared by multiple callers.

3.3 Analysis Domain

The states of our abstract domain have the form of $\overline{\text{State}} = \overline{H} \times \overline{V} \times \overline{F}$ where $\overline{H}$ is a state of the heap analysis, $\overline{V}$ is a state of the value analysis, and $\overline{F}$ is a state of the declaration analysis. The heap analysis tracks the shape of the heap and approximates concrete heap locations with heap identifiers (HId), while the value analysis tracks values on heap identifiers. While the heap analysis and value analysis need to interplay, the declaration analysis is independent from both.

Declaration Analysis

Declaration analysis is necessary, because in PHP and other dynamic languages, the names of functions, classes, and constants are bound to concrete definitions during runtime. The analysis thus needs to track these definitions. A state of a declaration analysis $\overline{F}$ is a set of class, function, and constant declarations and lattice operators of the analysis are $\langle \overline{F}, \subseteq, \cup, \cap \rangle$.

▶ **Example 3.** Consider the following PHP code:

```
1
2  if ($_GET[1]) {
3    class A {
4      public $a = 2;
5      function f($p) { return $p + 1;}
6    }
7  } else {
8    class A {
9      public $a = −1;
10     function f($p) { return $p − 1;}
11   }
12 }
13 $x = new A();
14 $y = $x−>f($x−>a); // $y can be −2, 0, 1, 3
```

Since the condition at line (1) is statically unknown, the declaration analysis computes that both declarations of class `A` can be used at line (12). Consequently, the call at line (13) can be done with two possible arguments and has two possible callees resulting in four possible results.

Heap Analysis

In PHP and other dynamic languages, variables as well as array indices and object properties need not be declared and can be accessed with arbitrary expressions, which can yield statically unknown values. If a specified variable, index, or object property exists, it is overwritten; if not, it is created.

To be able to capture this semantics, the heap analysis approximates arrays, objects, array indices, object fields, and even variables[2] with heap identifiers and the heap analysis can create new heap identifiers both during assignment and join operation. Whenever a new heap identifier is created, it is initialized with an existing heap identifier that stores values from statically unknown assignments to the new identifier that could happen before the creation.

Creation of new heap identifiers corresponds to *materialization* in shape analysis [20]. The *summary* heap identifiers summarize all the heap elements that could be updated by statically unknown assignments and have not been materialized yet[3]. When there is a need to distinguish a heap element from other heap elements summarized by the same summary heap identifier, a new heap identifier is materialized from the summary identifier. This happens, e.g., when an array index is assigned for the first time with a statically known target. In this case, the array index is approximated by the summary heap identifier representing all indices that could be updated only by statically unknown assignments in the pre-state and by the newly materialized heap identifier in the post-state. Materializations are defined as a set of pairs of heap identifiers $\mathrm{Mat} = \mathcal{P}(\mathrm{HId} \times \mathrm{HId})$. The meaning of a single materialization is that the first heap identifier from the pair is materialized from the second, summary heap identifier.

Note that materialization makes the naming scheme of the heap analysis flow-dependent – depending on the program location, a concrete heap element can be approximated by different heap identifiers. This makes an interplay of the heap analysis and the value analysis more

[2] Variables are treated as indices of a special associative array representing the symbol table.
[3] Heap elements that have not been assigned by any assignment are summarized by a special heap identifier `uninit`.

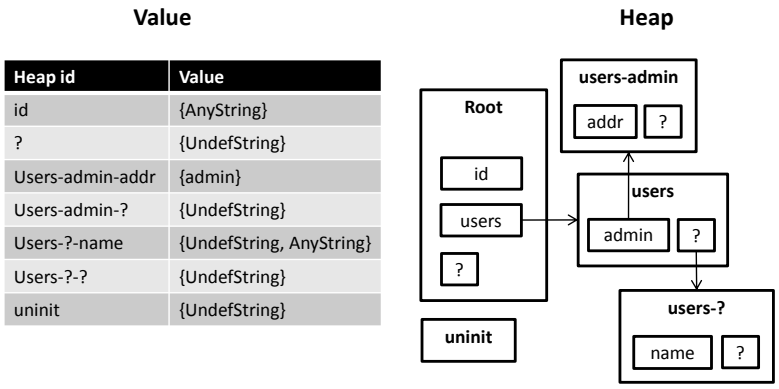

Figure 5 The heap and string part of the value component of the state after the update at line 23 in Fig. 1.

challenging. Since the value analysis tracks values on heap identifiers, materializations, which change the naming scheme, need to be applied also to value analysis. Later, we define this application using the standard abstract interpretation interface of the value analysis. This makes it possible to update the state of the value analysis automatically, without modifying the value analysis ad-hoc. That is, any value analysis that complies with the standard abstract interpretation interface can be used.

▶ **Example 4.** Fig. 5 shows the heap and value component of the analysis state after the update at line 23 in Fig. 1. In the following, we use the heap domain developed in [11] and the set domain as a value domain. Note that the value domain tracks values just over these heap identifiers that can contain values. Other heap identifiers are present only in the heap domain.

The heap component of the state contains heap identifier `Root` representing the array corresponding to the symbol table and heap identifier `uninit` representing the uninitialized heap elements. The symbol table array contains three heap identifiers (`id`, `users`, and `?`), which represent program variables (`$id` and `$users`) and statically unknown variables. For heap identifier `id`, value analysis tracks the value `AnyString`, while heap identifier `users` is present only in the heap domain and points to another array. Heap identifier `users-admin` represents index `$users[admin]`, while heap identifier `users-?` represents statically unknown indices of array `$users`. Both heap identifiers represent arrays corresponding to the next dimensions of array `$users`. Finally, heap identifiers `users-admin-addr`, `users-admin-name`, `users-admin-?`, `users-?-name`, and `users-?-?` represent indices of these arrays. Since these heap identifiers store values, they are tracked by the value analysis.

We assume that heap analysis is provided with lattice operators $\langle \overline{H}, \sqsubseteq_{\overline{H}}, \sqcup_{\overline{H}}, \sqcap_{\overline{H}} \rangle$. Operator $\sqsubseteq_{\overline{H}}$ specifies a partial order, $\sqcup_{\overline{H}}$ is the join operator, and $\sqcap_{\overline{H}}$ is the meet operator. The semantics of heap analysis is given by transfer function $[\![\cdot]\!]_{\overline{H}} : \overline{H} \mapsto \overline{H}$.

Moreover, we assume that the heap analysis provides function read : $AE \mapsto \mathcal{P}(\text{HId})$ for reading data from the heap. The function returns a set of heap identifiers identified by given *access expression*. Access expression is obtained from IR nodes of type variable^V, property-use^V, and index-use^V. In the case of variable^V, access expression is just the set of values, in the case of property-use^V, and index-use^V, it is a sequence of sets of values. Each set from the sequence contains values that can be used to access the corresponding dimension of an array or the corresponding object in the object reference chain. That is,

access expressions can represent multi-dimensional updates. This is necessary in order to model semantics of non-decomposable multi-dimensional updates [11].

▶ **Example 5.** Consider reading the values stored at index `$users[10]['name']` from the state depicted in Fig. 5. The access expression for the index is $\{users\}$ $\{10\}$ $\{'name'\}$. Calling the read function provided by the heap component with this access expression as argument returns the heap identifier `users-?-name`. This heap identifier is then used to get the resulting values from the value domain. The value domain returns values `UndefString` and `AnyString` meaning that the index `$users[10]['name']` can be uninitialized and can contain statically unknown string value.

Similarly, when reading the index `$users[$_GET[1]]['name']`, the access expression is $\{users\}\{*\}\{'name'\}$, the read function returns heap identifiers `users-?-name` and `users-admin-name`, and the subsequent call to the value domain returns values `UndefString`, `AnyString`, and `'addr'`.

We additionally assume that the heap analysis provides function joinToValue : $\overline{H} \times \overline{H} \mapsto$ Mat $\times$ Mat. This function takes the heap parts of analysis states to be joined as arguments and for each joined analysis state, it returns materializations of the heap identifiers created when performing the join operation.

Finally, we assume that heap analysis provides function assignToValue : $\overline{H} \times$ AE $\mapsto$ Mat $\times \mathcal{P}$(HId) $\times \mathcal{P}$(HId). This function takes an analysis state before the assignment and the access expression identifying the target of the expression as arguments and returns a triple: (i) materializations of heap identifiers created when performing the assignment, (ii) the heap identifiers representing heap elements that certainly must be updated, and (iii) the heap identifiers representing heap elements that only may be updated.

Value Analysis

The states of the value analysis have a form of $\overline{V} = \overline{V_1} \times \overline{V_2}$ where $\overline{V_1}$ is a state of the value analysis in the first phase and $\overline{V_2}$ is a state of the value analysis in the second phase (end-user analysis). The first phase of the analysis modifies the first component of the state V_1, the second phase of the analysis modifies the second component of the state V_2.

The user of the framework can define both the value analysis in the first phase and the value analysis in the second phase. However, since values that are used to compute control-flow and targets of data accesses are computed in the first phase, the user is more constrained in the first phase.

Second phase

The domain for the second phase tracks information over heap identifiers and it is provided with lattice operators $\langle \overline{V_2}, \sqsubseteq_{\overline{V_2}}, \sqcup_{\overline{V_2}}, \sqcap_{\overline{V_2}} \rangle$, transfer function $[\![\cdot]\!]_{\overline{V_2}} : \overline{V_2} \mapsto \overline{V_2}$, and widening operator $\triangledown_{\overline{V_2}}$.

First phase

In the first phase, the value analysis tracks values of PHP primitive types over heap identifiers:

$\overline{V_1} = \text{HId} \mapsto \overline{\text{Value}_1}$
$\overline{\text{Value}_1} = \overline{\text{Undef}} \times \overline{\text{Null}} \times \overline{\text{Bool}} \times \overline{\text{Num}} \times \overline{\text{String}}$

Since PHP has dynamic type system – variables, array indices, and object properties do not have declared types, and they can store values of different types depending on context –, $\overline{\text{Value}_1}$ can store values of all primitive types.

To define the value analysis of the first phase, the user of the framework must for each component $\overline{C}$ of $\overline{\text{Value}_1}$ provide the framework with lattice operators $\langle \overline{C}, \sqsubseteq_{\overline{C}}, \sqcup_{\overline{C}}, \sqcap_{\overline{C}} \rangle$, transfer function $[\![\cdot]\!]_{\overline{C}} : \overline{C} \mapsto \overline{C}$, and widening operator $\triangledown_{\overline{C}}$. The lattice operators for $\overline{\text{Value}_1}$ are defined component-wise. Moreover, for each pair $(\overline{C_1}, \overline{C_2})$ of components of $\overline{\text{Value}_1}$, functions $\text{Conv}_{\overline{C_1 C_2}} : \overline{C_1} \mapsto \overline{C_2}$ and $\text{Conv}_{\overline{C_2 C_1}} : \overline{C_2} \mapsto \overline{C_1}$ must be provided. These functions are used to model type conversions, which are ubiquitous in dynamic languages.

It should be noted that even though value analysis in the first phase is defined independently of heap analysis, which simplifies its definition, intricate value semantics of PHP makes the definition inherently complex. The framework thus provides default implementations of all components of $\overline{\text{Value}_1}$ including transition functions for PHP native operators and library functions. For the default implementation of the numeric component, we used the interval domain. For the default implementation of the string component, we used the domain based on sets of strings. Its lattice structure is $\langle \mathcal{P}(\text{String}), \subseteq \rangle$ where String are concrete strings. To make the height of the lattice finite and thus guarantee termination, the size of sets is limited by a constant; value AnyString represents the sets of larger sizes.

▶ **Example 6.** This example illustrates the states of $\overline{\text{Value}_1}$ with the default implementation of its components.

The abstract value $(\bot, \bot, \text{AnyBool}, \bot, \bot)$ represents concrete values `true` and `false` of type Boolean, the abstract value (undef, $\bot, \bot$, true, {"foo", "bar"}) represents the following concrete values: uninitialized value, the Boolean `true`, the string `"foo"`, and the string `"bar"`.

3.4 Lattice Order and Meet

The lattice order $\sqsubseteq_{\overline{\text{State}}}$ and meet operator $\sqcap_{\overline{\text{State}}}$ for analysis states are defined component-wise:

$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqsubseteq_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) \iff h_1 \sqsubseteq_{\overline{H}} \overline{h_2} \wedge \overline{v_1} \sqsubseteq_{\overline{V}} \overline{v_2} \wedge \overline{f_1} \subseteq \overline{f_2}$$
$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqcap_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) = (\overline{h_1} \sqcap_{\overline{H}} \overline{h_2}, \overline{v_1} \sqcap_{\overline{V}} \overline{v_2}, \overline{f_1} \cap \overline{f_2})$$

3.5 Applying Materializations to Value Analysis

Materializations allow the heap analysis to create new heap identifiers during the assignment and join operations. As discussed in Sect. 3.3, materializations change the naming scheme of the heap analysis; since value analysis tracks values of heap identifiers, these changes must be applied also to the value domain. This is carried out by function applyMaterializations : $(\overline{V} \times \text{Mat}) \mapsto \overline{V}$ that applies materializations to a state of the value analysis:

$$\text{applyMaterializations}(\overline{v}, M) = \overline{v_n} \text{ where } M = \{(t_1, s_1), (t_2, s_2), ..., (t_n, s_n)\},$$
$$\overline{v_0} = \overline{v}, \forall j \in [1..n] : \overline{v_j} = [\![t_j = s_j]\!]_{\overline{V}}(\overline{v_{j-1}})$$

3.6 Join and Widening

The join of two facts is defined as the set of all facts that are implied independently by any. The join and widening of two states (h_1, v_1, f_1) and (h_2, v_2, f_2) are defined as follows:

$$(\overline{h_1}, \overline{v_1}, \overline{f_1}) \sqcup_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}, \overline{f_2}) = (\overline{h_1} \sqcup_{\overline{H}} \overline{h_2}, \overline{v_1'} \sqcup_{\overline{V}} \overline{v_2'}, \overline{f_1} \cup \overline{f_2})$$
$$(\overline{h_1}, \overline{v_1}) \triangledown_{\overline{\text{state}}} (\overline{h_2}, \overline{v_2}) = (\overline{h_1} \sqcup_{\overline{H}} \overline{h_2}, \overline{v_1'} \triangledown_{\overline{V}} \overline{v_2'}, \overline{f_1} \cup \overline{f_2})$$
$$(\overline{m_1}, \overline{m_2}) = \text{joinToValue}(\overline{h_1}, \overline{h_2})$$
$$\overline{v_1'} = \text{applyMaterializations}(\overline{v_1}, \overline{m_1})$$
$$\overline{v_2'} = \text{applyMaterializations}(\overline{v_2}, \overline{m_2})$$

The declaration and heap parts of input states are joined independently on other parts. To perform the join of value parts, heap analysis provides value analysis with materializations of heap identifiers in each joined state. Then, the materializations are applied to the value components of joined states and finally, the updated value parts are joined. Note that the latter two operations are done just by means of standard abstract interpretation interface provided by value analysis.

▶ **Example 7.** Fig. 6 shows joining value and heap components of two states $(\overline{v_1}, \overline{h_1})$ and $(\overline{v_2}, \overline{h_2})$. For brevity we omit declaration components.

First, the heap components of analysis states are joined. For the first state to be joined, heap analysis materializes heap identifiers `arr-1-3`, `arr-1-?`, and `arr-?-?` from the heap identifier `uninit` representing undefined heap identifier. That is, there were no statically-unknown assignments that could update the materialized identifiers. Application of materializations to the value component of the first analysis state to be joined thus just adds the materialized identifiers and initializes them with value `UndefString`.

For the second state, heap analysis materializes heap identifiers `arr-1-?`, `arr-1-2`, and `arr-1-3`. Since there was statically-unknown assignment that could update the latter identifier, this identifier is materialized from the identifier `arr-?-3` representing the target of this statically-unknown assignment. Application of materializations to the value component of the second analysis state to be joined thus initializes the identifier `arr-1-3` with values `UndefString` and `'second'`.

Finally, the value components of analysis states after applying materializations $\overline{v_1'}$ and $\overline{v_2'}$ have the same set of heap identifiers and can be joined independently on the heap components.

3.7 Transfer Functions

For each kind of node in the intermediate representation, a transfer function maps an abstract state before the node to an abstract state after the node.

We describe the transfer function for the node assign$^V[l^V, r^V]$, where both parameters l^V and r^V are nodes of type variable^V, property-use^V, or index-use^V. Each of these nodes allows getting an access expression, which provides heap analysis information necessary for accessing heap identifiers representing heap locations stored in the node. The access expression for the left-hand side of the assignment l^V is l^V.AE, the access expression for the right-hand side of the assignment r^V is r^V.AE.

To define the transfer function, we first define function strongUpdate : $\overline{V} \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId}) \mapsto \overline{V}$. The first parameter is a state of value analysis that is being updated, the second parameter is the set of heap identifiers that are updated, and the last parameter is

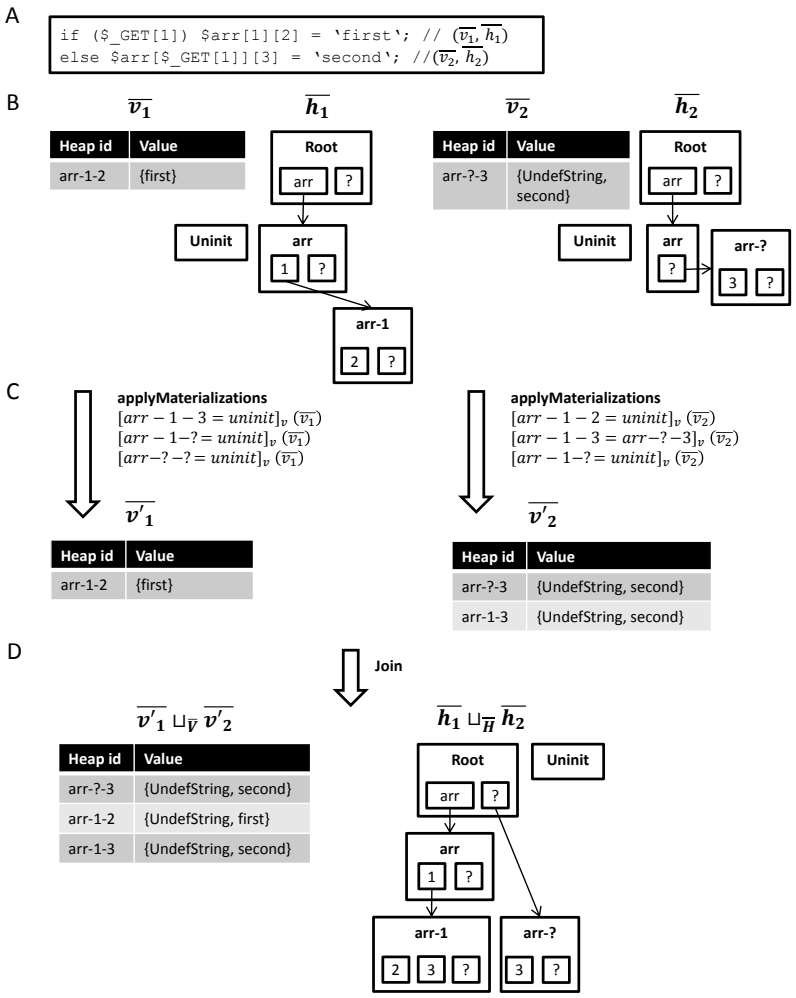

Figure 6 Joining value and heap components of two states $(\overline{v_1}, \overline{h_1})$ and $(\overline{v_2}, \overline{h_2})$. The corresponding code (A), value and heap components of joined states (B), applying materializations to value components of states to be joined (C), result of the join (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

the set of heap identifiers representing new values:

$$\text{strongUpdate}(\overline{v}, T, S) = \overline{v_n} \ \textit{where}\ T = \{t_1, t_2, ..., t_n\}, \overline{v_0} = \overline{v}$$
$$\forall j \in [1..n] : \overline{v_j} = \bigsqcup_{s \in S} [\![t_j = s]\!]_{\overline{V}}(\overline{v_{j-1}})$$

Next, we define function $\text{weakUpdate} : \overline{V} \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId}) \mapsto \overline{V}$:

$$\text{weakUpdate}(\overline{v}, T, S) = \bigsqcup_{t \in T, s \in S} \overline{v} \sqcup_{\overline{V}} [\![t = s]\!]_{\overline{V}}(\overline{v})$$

While after strong update, heap identifiers can have just new values, after weak update, they either can have the original values or the new ones [18]. This effect is approximated by joining the analysis state before the update with the analysis state after the update.

The transfer function for updating the state $(\overline{h}, \overline{v})$ with $\text{assign}^V[l^V, r^V]$ is defined as:

$[\![\text{assign}^V[l^V, r^V]]\!]_{\overline{\text{state}}}(\overline{h}, \overline{v}, \overline{f}) = ([\![l^V.\text{AE} = r^V.\text{AE}]\!]_{\overline{H}}(\overline{h}), \overline{v'''}, \overline{f})$
$(m, u_{must}, u_{may}) = \text{assignToValue}(\overline{h}, l^V.\text{AE})$
$\overline{v'} = \text{applyMaterializations}(\overline{v}, m)$
$\overline{v''} = \text{strongUpdate}(\overline{v'}, u_{must}, \text{read}(\overline{h}, r^V.\text{AE}))$
$\overline{v'''} = \text{weakUpdate}(\overline{v''}, u_{may}, \text{read}(\overline{h}, r^V.AE))$

The transfer function for the heap part of the state is defined by the heap domain itself, and it is not influenced by the value domain. To define the transfer function for the value part of the state, the heap domain provides the value domain with necessary information via function assignToValue. This information consists of: (1) m – information necessary to materialize the heap identifiers that were defined by the assignment (2) heap identifiers representing the heap elements that are certainly targets of the assignment, and (3) heap identifiers representing the heap elements that may be targets of the assignment.

Then, the materializations are applied to the value domain. Finally, the heap identifiers that are certainly targets of the assignment are strongly updated with values of the heap identifiers read from the right-hand side of the assignment, and the heap identifiers that only may be targets of the assignment are weakly updated. The same way as in the case of the join operation, all these updates are performed just by means of the transfer function for the assignment provided by value analysis.

▶ **Example 8.** Fig. 7 illustrates the transition function for the assignment at line 24 in Fig. 1 (`$users[$id]['addr'] = $_GET['addr']`). First, the access expressions for the source and the target of the assignment are obtained from the corresponding IR nodes. For the source of the assignment, the access expression is $\{\_GET\}\{addr\}$, for the target of the assignment, the access expression is $\{users\}\{AnyString\}\{addr\}$. Note that in the latter case, the value for the second dimension of the access is specified by the variable `$id`.

Second, the access expressions are used to specify the update. During the update, the heap component materializes the heap identifier `users-?-addr` and this change is propagated to the value component via the function applyMaterializations. Note that since there have not been any statically unknown assignments that could update this heap identifier, it is materialized from the identifier `uninit` representing undefined values. That is, the identifier `users-?-addr` is added to the value component and initialized with `UndefString`.

Finally, the heap component specifies that identifiers `users-?-addr` and `users-admin-addr` are weakly updated and the update is propagated to the value component. Since the target of the assignment is not statically known, no heap identifiers are strongly updated.

3.8 Summary Heap Identifiers

Value analyses are designed to track information on local variables, while we use them to track information on heap identifiers, which can represent many concrete heap locations – summary identifiers [9]. For an example of summary identifiers, consider heap identifiers representing targets of statically unknown assignments and heap identifiers representing a single allocation-site in that many concrete heap locations can be allocated. While value analysis can treat heap identifiers that represent a single heap location exactly the same way as local variables, for summary identifiers, it must take into account that they represent more heap locations.

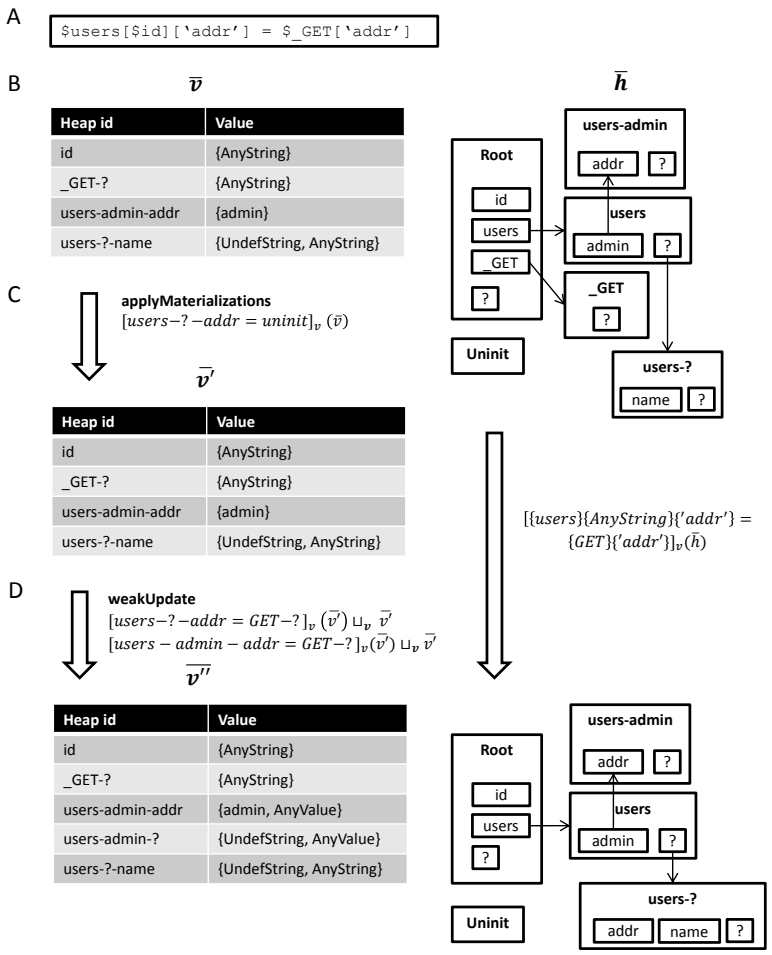

Figure 7 Transfer function for the assignment. The code of the assignment (A). The value and the heap component $(\overline{v}, \overline{h})$ of the state before the assignment (B). Applying materialization of the identifier `users-?-addr` to the value component of the state (C). The value component $\overline{v''}$ and the heap component of the state after the assignment (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

First, summary heap identifiers must be always weakly updated. In our framework, heap analysis has to take this into account in function assignToValue, which defines identifiers that are weakly and strongly updated by the assignment. This is enough for non-relational value domains – these value domains can otherwise treat summary heap identifiers the same way as local variables.

However, in the case of relational value domains, it is additionally necessary to treat differently assignments from summary heap identifiers. Consider the code:

```
1  1
2  $a = $users[$_GET[1]];
3  $b = $users[$_GET[2]];
4  if ($a != $b) {...}
```

Our heap analysis represents both `$users[$_GET[1]]` and `$users[$_GET [2]]` by the same summary heap identifier `users-?`. Our technique would thus abstract the semantics of assignment at line (1) as $[\![a = users{-}?]\!]_{\overline{V}}$ and the semantics of assignment at line (2) as

$[\![b = users{-}?]\!]_{\overline{V}}$. If v werte a relational domain, the analysis would relate both identifiers `a` and `b` with the summary identifier `users-?` and incorrectly infer that the `if-then` branch can never be reached. This problem was studied by Gopan [9] et. al., who showed that it is wrong to correlate summarized identifiers with non-summarized ones and they proposed a way to extend existing relational domains to deal with this problem.

In our framework, the value domain in the first phase is non-relational and all value domains for end-user analyses that we have implemented so far are also non-relational. To use relational value analyses, these analyses need to be extended to summary dimensions as described by Gopan [9] and the heap analysis has to additionally provide the framework with the information which heap identifiers are summaries.

3.9 Soundness

Our analysis framework allows for defining sound analyses. If the semantics of heap analysis, the semantics of value analysis, and the semantics of declaration analysis plugged into our framework are sound, the semantics of the resulting composed analysis is sound as well. In the following, we will state the fundamental assumptions on value and heap analyses[4]. The traditional soundness argument in abstract interpretation is:

▶ **Definition 9** (Soundness of analysis semantics)**.** Given a set of abstract states $\overline{S}$, abstract semantics $[\![\cdot]\!]_{\overline{S}}$, a set of concrete states S, concrete semantics $[\![\cdot]\!]_S$, and concretization function $\gamma : \overline{S} \mapsto \mathcal{P}(S)$, the abstract semantics $[\![\cdot]\!]_{\overline{S}}$ is sound with respect to the concrete semantics $[\![\cdot]\!]_S$ iff for each statement st and analysis state $\overline{s} \in \overline{S}$ it holds:

$$([\![st]\!]_{\overline{S}}(\overline{s}) = \overline{s}' \wedge [\![st]\!]_S(\gamma(\overline{s})) = s') \implies s' \subseteq \gamma(\overline{s}')$$

Hence, to prove the soundness of the analysis semantics, it is necessary to define the structure of concrete states, their semantics, concretization function, and then prove that it satisfies proposition of Def. 9.

The soundness argument is based on the assumption that the heap semantics and the value semantics are sound. While for the value semantics, the soundness can be specified just using Def. 9 and we can thus use any sound value analysis in our framework, for the heap semantics, we must pose further assumptions.

First, we assume that function read provided by heap analysis complies with the semantics of concrete dereferencing. That is, for each abstract state and each access expression, the heap identifiers returned by function read must represent all concrete locations given by dereferencing using the access expression in all the concretizations of the abstract state.

Next, we assume that the updates given by semantics function assignToValue are sound with respect to the semantics of concrete dereferencing. That is, for the left hand site of assignment, the heap identifiers representing updates given by function assignToValue and an access expression in an abstract state must represent all concrete heap locations R given by dereferencing using the access expression in all the concretizations of the abstract state. Moreover, all the heap identifiers that are in the strong-update set (u_{must}) must exactly represent all the heap locations in set R and all the other heap identifiers that represent the sets of heap locations with non-empty intersection with R must be in the may-update set (u_{may}).

[4] We will omit the declaration analysis. It needs not interplay with the other analyses and can be treated completely separately.

Finally, we assume that the materializations produced by heap analysis are coherent with respect to the modifications of heap analysis. That is, (1) in the post-state, the heap identifiers that are not sources of materializations must represent the same concrete heap locations as in the pre-state, (2) for each heap identifier that is materialized and its source it must hold that in the post-state each of them represents the subset of the heap locations represented by the source of the materialization in the pre-state, and (3) in the post-state both heap identifiers together must represent all the heap locations represented by the source of the materialization in the pre-state.

Note that we do not require different heap identifiers to represent non-overlapping portions of concrete heap. That is, there can exist two different heap identifiers with *overlapping concretizations*, i.e., there exists a concrete heap location approximated by both heap identifiers. This allows using heap analyses modeling the semantics of assignment by reference more precisely [11]. Consider, e.g., the statement `$a = &$b`. In the concrete semantics, the statement makes `$a` and `$b` pointing to the same heap location. We allow heap analysis to model this concrete location by more heap identifiers with overlapping concretizations – e.g., heap identifier i_1 for variable `$a` and heap identifier i_2 for variable `$b`. To be sound, heap analysis must update these heap identifiers coherently – e.g., if heap identifier i_1 is updated, heap identifier i_2 is updated as well. This is guaranteed by the soundness of updates stated above. Heap identifiers with overlapping concretizations can enable more strong updates. Consider the following example:

```
1  if ($_GET['INPUT']) $a = &$b;
2  else $a = &$c;
3  $a = 1;
```

There are two concrete heap locations in this example. If heap analysis uses less than three heap identifiers to represent these concrete locations, it must perform weak update at line 3. In case of three heap identifiers, their concretizations must overlap; it allows heap analysis to perform strong update on the heap identifier for `$a` and weak updates of those for `$b` and `$c`.

4 Evaluation

To evaluate the precision and scalability of our framework, we used it to implement static taint analysis and we applied it to the NOCC webmail client[5] and a benchmark application comprising of a fragment of the myBloggie weblog system[6], with a total of over 16 kLoC. The benchmark application contains 13 security problems; the number of problems contained in the webmail client is not known.

We compared the results of our analysis with Pixy [15] and Phantm [17], the state-of-the-art tools for security analysis and error discovery in PHP applications. Both these tools compute control-flow of analyzed applications, model PHP data structures, and perform value analysis. Both these tools detect accesses to uninitialized elements. In addition, Phantm detects type mismatch errors and Pixy detects taint errors, i.e., flows of sensitive data to critical commands. Our analysis detects both type of errors.

Tab. 2 shows the summary of results. Out of 13 errors in the benchmark application, 8 errors were accesses to uninitialized elements and 5 errors were taint errors. Since Pixy is not designed to detect taint errors we did not use taint errors to assess the error coverage

[5] `http://nocc.sourceforge.net/`
[6] `http://mybloggie.mywebland.com/`

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).

	myBloggie				**NOCC 1.9.4**			
Lines	648				15,605			
	W	**C**	**F**	**T**	**W**	**C**	**F**	**T**
Our framework	16	100	19	0.9	34	NA	62	84
Pixy	16	69	44	0.6				NA
Phantm	43	38	93	2.5	426	NA	NA	90

for Pixy. The table shows that the analysis defined using our framework outperforms the other tools both in error coverage and number of false positives when analyzing the benchmark application. As to the analysis of NOCC, while Pixy was even not able to analyze the application, Phantm reported a huge number of alarms, which together with a high false-positive rate made its output almost useless[7].

Our analysis discovered all 13 problems in myBloggie. One of the false alarms reported by our analysis is caused by imprecise modeling of the built-in function `date`. Our analysis 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, our analysis 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.

Our analysis found three previously unknown vulnerabilities in the NOCC email client and ten other problems (e.g., calling a function with an argument that is not declared and superfluous implicit conversions). False-positive alarms were caused by imprecise modeling of PHP built-in functions, path-insensitivity of the analysis, and using non-relational value domains.

5 Related Work

The present work builds on a large body of work on static program analysis of dynamic languages. The pioneering works [12, 30, 26, 27] omit modeling some of the important parts of the analyzed languages. The unmodeled parts include references, dynamic accesses to associative arrays, and object-oriented features.

Pixy [16] performs security taint analysis of PHP programs and provides information about the flow of tainted data. Pixy performs a flow-sensitive, interprocedural, and context-sensitive data flow analysis along with literal and alias analysis to achieve precise results. Its main limitations include an incomplete support for statically-unknown updates to associative arrays, ignoring classes and the `eval` command, omitting type inference, and a limited support for handling file inclusion and aliasing. Alias analysis introduced in Pixy incorrectly models aliasing when associative arrays and objects are involved.

[7] Because of a huge number of alarms reported by Phantm, we assessed its false-positives rate just for myBloggie, not for NOCC.

Andromeda static taint analyzer [24] 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. Especially for dynamic languages, the control-flow of the application can depend on other kind of information which is then not available. To reduce this problem, Andromeda uses F4F [21], which reduces the amount of information that is not known statically.

Phantm [17] 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, 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.

TAJS [14] is a JavaScript static program analysis infrastructure that infers type information. To gain precise results, the analysis is context-sensitive and precisely models intricate semantics of JavaScript, including prototype objects and associative arrays, dynamic accesses to these data structures, and implicit conversions. It tackles the problem that dynamic features of JavaScript make it impossible to construct control-flow before static analysis by constructing control-flow on-the-fly during the analysis. Since TAJS models JavaScript semantics precisely, it has been successfully used to enable additional analyses. In [4, 5], the TAJS program analysis infrastructure is used to build a tool for refactoring JavaScript programs and in [13] TAJS is used to enable technique of statically resolving `eval` constructs. However, TAJS combines heap and value (type) analysis ad-hoc, which results in intricate lattice structure and transfer functions. Next, TAJS assumes that updates to multi-dimensional arrays and objects can be decomposed to updates of length one. While this is true for JavaScript, this assumption leads to loss of precision in the case of some other dynamic languages such as PHP and Perl.

Since the excess of information that are only available at runtime pose a major problem to static analysis, several techniques have been developed that try to enable static analysis of dynamic languages by making this information statically available prior to static analysis. F4F [21] focuses on static taint analysis of web applications that use frameworks. They use a semi-automatically generated specification of framework-related behaviors to reduce the amount of statically-unknown information, which arises, e.g., from reflective calls. Phantm [17] reduces the number of information that static analysis must compute and possibly overapproximate by first executing the application, collecting this information and then invoking static analysis from a particular runtime state. Wei et. al. [28] reduce the number of statically-unknown information in JavaScript by using a technique of blended static analysis [2]. They first execute a test suite and for each test they record its execution trace. Then, for each execution trace, they extract its call graph, types of created objects, and dynamically generated code and perform static analysis of the application with using this information. Finally, they combine solutions from different execution traces into a single solution for the application.

Recently, there has flourished a rich body of work on precise and sound points-to analysis for dynamic languages. Sridharan et. al. [22] present static flow-insensitive points-to analysis for JavaScript modeling objects in JavaScript using associative arrays that can be accessed by arbitrary expressions. 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 being the variable. Thus, the correlation between the update and store is preserved. Wei et al. [29] present points-to analysis for JavaScript. Their analysis is partially flow-sensitive – it stores points-to information for every CFG segment with a single state-update statement. Next, their analysis is context-sensitive – to reflect the fact that properties can be added to objects at runtime, it uses a receiver object, its chain of prototype objects, its local properties and their object values as a context. This makes it possible to differentiate between two calls received by the object with the same creation site but different properties. Finally, to perform more strong updates in case of property-writes they use access path edges in their points-to representation. For a property-write (e.g., `$x->p = $y`) where the dereferenced variable (`$x`) points to more objects, weak-updates of properties in these objects (`p`) are performed. However, the access path edge (`<x, p>`) is strongly updated. If the property is read and there exists a corresponding access path edge, it is used instead of points-to edges (for `$z = $x->p` the access path edge `<x, p>` is used and just objects pointed-to by variable `$x` are read). In our previous work [11], we presented points-to analysis for PHP modeling associative arrays that could be accessed using arbitrary expressions. Additionally, our analysis precisely models the semantics of PHP explicit aliases and the semantics of multi-dimensional updates – in PHP or Perl, updates create indices if they do not exist and initialize them with empty arrays if needed; on contrary, read accesses do not.

While heap and value static analyses have been studied mainly as orthogonal problems, to support verification of real programs, they usually need to be combined together [6, 25]. Since in dynamic languages, data structures can be dynamically accessed with arbitrary expressions, this problem of combining heap and static value analysis is particularly relevant in this domain.

Clousot [3] preprocesses the program applying heap analysis, and uses a value numbering algorithm to compute under-approximation of must-alias to replace heap accesses with heap identifiers. Value analysis then tracks values of variables and also of the heap identifiers. While the approach allows for using arbitrary value analysis, it only allows for using specific heap analysis, which cannot use the information provided by value analysis, and the technique is not sound.

Miné et. al. [19] combine type based pointer analysis and numeric value analyses in a generic way. The pointer analysis models pointer arithmetic, union types and records of stack variables in C programs. The general limitation of this technique is that it relies on type based heap analysis, which is too coarse for many applications. In particular, their technique does not support summary nodes and dynamic allocation.

Fu [8] combines numeric value analysis and points-to analysis. His method uses points-to analysis to partition possibly infinite set of heap references into a finite set of abstract locations (heap identifiers) and use value analysis to track values of variables and also of heap identifiers. The method is both generic – it allows for reusing existing analyses as black-boxes – and automatic – it does not require any annotations specific to a particular heap and value analysis to be provided. The fundamental limitation of the technique is that it relies on flow-independent naming scheme for points-to analysis. That is, a concrete reference is always mapped to the same abstract location independently of program location. On one hand, this assumption allows the technique to assume that change of the heap component of the analysis state has no effect on the value component of the state and that two states can

be joined component-wise. On the other hand, this assumption poses a substantial limitation to modeling of adding new object fields and array indices using statically-unknown updates. To illustrate the limitation, consider that a statically-unknown index of an empty array `$a` is updated (`$a[rand()]=..`). At this point, points-to analysis must represent all concrete indices of the array with a single abstract location h. Next, if a concrete index of the array, e.g., `$a[1]`, is updated (`$a[1]=..`), the analysis must still represent the index `$a[1]` with h and thus cannot distinguish this index from other indices in `$a`. That is, a statically unknown update makes the updated array (object) index-insensitive (field-insensitive) for all indices (fields) added after the update. As both modeling statically-unknown updates and field-sensitivity of heap analysis is crucial for static analysis of dynamic languages [23, 29], the assumption of flow-independent naming scheme is too limiting in this context.

Ferrara [6] introduced the concept of substitutions overcoming the limitation of flow-indepenent naming scheme when combining value and heap analysis. Substitutions allow heap analysis to materialize and summarize abstract locations, i.e., to replace a single abstract location in the pre-state with more abstract locations in the post-state and to replace more abstract locations in the pre-state with a single abstract location in the post-state. Ferrara defined how the analyses are composed when the substitutions are given and showed the assumptions on the heap and the value analyses in order to make their composition sound. However, his work cannot be directly applied in the context of dynamic languages. First, it does not model dynamically added fields and indices to objects and arrays, which is essential for dynamic languages. Then, Ferrara allows only heap analyses with non-overlapping heap identifiers. As we explain in Section 3.9, some heap analyses developed for dynamic languages use overlapping heap identifiers to perform more strong updates. Moreover, his work does not allow heap analyses to explicitly specify which updates are strong and which updates are weak thus reducing the precision of the composed analysis. In our work we focused specifically on heap analyses for dynamic languages overcoming these limitations.

6 Conclusion

In this paper, we presented a framework for static analysis of dynamic languages, in particular PHP applications.

The framework employs a two-phase analysis architecture – in the first phase, the dynamic constructs present in the analyzed code are resolved, while the analysis in the second phase can proceed in a way similar to a one for a language without dynamic features. This way, the framework provides a developer with high-level API for implementing various kind of analyses upon the code without the need to cope with dynamic features of the language. To allow resolving dynamic features, the framework combines heap, value, and declaration analyses. We described the necessary requirements on these analyses and the way these analyses are composed together generically and soundly. That is, our framework allows for combining various heap and value analyses while guaranteeing that if the analyses being composed are sound, the composed analysis is sound as well.

The framework is provided with default implementations of heap analyses and first phase analyses. To demonstrate usefulness of our framework, we implemented taint analysis of PHP application and applied it on real PHP application. We have shown that the tool is able to reveal real (previously unknown) security holes, while producing less false-positive alarms comparing to other state-of-the-art tools.

As for future work, we aim at improving the performance and precision of the analyzes provided by the framework especially in terms of scaling to large applications. In particular,

this includes the scalability improvements of the heap analysis, implementation of more choices of context-sensitivity, and devising precise modeling of more library functions. Next, we plan to enhance our implementation of security analysis and use the framework for implementing additional end-user analyses.

References

1 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

2 Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA'07*, pages 118–128. ACM, 2007.

3 Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS'10*, LNCS, pages 10–30. Springer-Verlag, 2011.

4 Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *OOPSLA'11*, pages 119–138. ACM, 2011.

5 Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In *OOPSLA'13*, pages 323–338. ACM, 2013.

6 Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI'05*, LNCS, pages 302–321. Springer-Verlag, 2014.

7 Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, May 2007.

8 Zhoulai Fu. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for java. In *VMCAI'05*, LNCS, pages 282–301. Springer-Verlag, 2014.

9 Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS'04*, LNCS, pages 512–529. Springer-Verlag, 2004.

10 David Hauzar and Jan Kofroň. WEVERCA. `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`, 2014.

11 David Hauzar, Jan Kofroň, and Pavel Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS'14*, EPTCS, pages 56–70. Open Publishing Association, 2014.

12 Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW'04*, pages 40–52. ACM, 2004.

13 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA 2012*, pages 34–44. ACM, 2012.

14 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS'09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

15 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP'06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

16 Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP'06*, pages 258–263. IEEE Computer Society, 2006.

17 Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *RV'10*, LNCS, pages 300–314. Springer-Verlag, 2010.

18 Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL'11*, pages 3–16, New York, NY, USA, 2011. ACM.

19 Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES'06*, pages 54–63. ACM, 2006.

20 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

21 Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA'11*, pages 1053–1068. ACM, 2011.

22 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP'12: Proceedings of the 26th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 435–458, Berlin, Heidelberg, 2012. Springer-Verlag.

23 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In *ECOOP'12*, LNCS, pages 435–458. Springer-Verlag, 2012.

24 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE'13*, LNCS, pages 210–225. Springer-Verlag, 2013.

25 Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE 2005*, LNCS, pages 227–236. Springer-Verlag, 2005.

26 Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07*, pages 32–41. ACM, 2007.

27 Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE'08*, pages 171–180. ACM, 2008.

28 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013*, pages 336–346. ACM, 2013.

29 Shiyi Wei and Barbara G. Ryder. State-sensitive points-to analysis for the dynamic behavior of javascript objects. In *ECOOP 2014*, volume 8586 of *LNCS*, pages 1–26. Springer Berlin Heidelberg, 2014.

30 Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06*. USENIX Association, 2006.