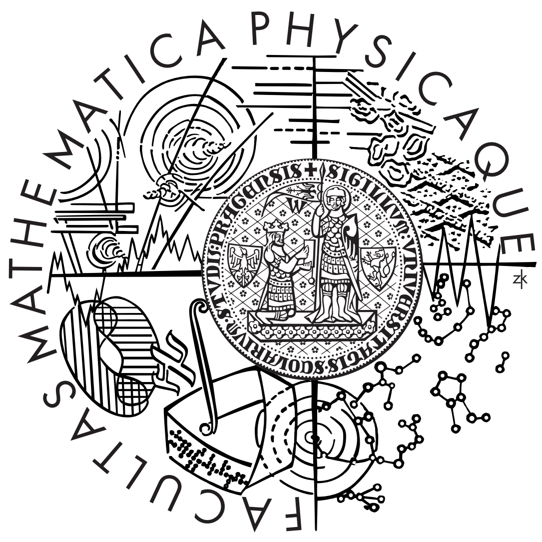
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Towards Static Analysis of Languages with Dynamic Features

David Hauzar

Department of Distributed and Dependable Systems
Advisor: Prof. František Plášil

I would like to thank all those who supported me in my doctoral study and in the research that resulted in this thesis. I very appreciate the help received from my advisor Prof. František Plášil and my co-advisor Dr. Jan Kofroň. I thank all my colleagues from the department for their continuous feedback and fruitful discussion. In particular, my thanks go to the rest of the formal methods group: Pavel Jančík, Jakub Daniel, Pavel Parízek, Ondřej Šerý, and Tomáš Poch. I would like to thank Viliam Šimko, with whom I collaborated on the FOAM method. Next, I would like to thank to students who joined WeVerca project and helped me with implementation. In particular, I would like to thank Mirek Vodolán and Pavel Báštecký.

This work and the related research was partially supported by the Grant Agency of the Czech Republic project 14-11384S and Charles University institutional funding (SVV-2014-260100, PRVOUK, SVV-2011-263312, and GAUK Project No 431011).

Last but not least, I am in debt to Ája and Jonáš. Their endless support and patience made this work possible.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, June 26, 2014 David Hauzar

Annotation

Title	*Towards Static Analysis of Languages with Dynamic Features*
Author	David Hauzar hauzar@d3s.mff.cuni.cz
Advisor	Prof. František Plášil plasil@d3s.mff.cuni.cz (+420) 221 914 266
Department	Department of Distributed and Dependable Systems Faculty of Mathematics and Physics Charles University in Prague Malostranské nám. 25, 118 00 Prague, Czech Republic

Abstract

Dynamic features of programming languages such as dynamic type system, dynamic method calls, dynamic code execution, and dynamic data structures provide the flexibility which can accelerate the development, but on the other hand they reduce the information that is checked at compile time and thus make programs more error-prone and less efficient. While the problem of lacking compile time checks can be partially addressed by techniques of static analysis, dynamic features pose major challenges for these techniques sacrificing their precision, soundness, and scalability. To tackle this problem, we propose a framework for static analysis that automatically resolves these features and thus allows defining sound and precise static analyses similarly as the analyzed program would not use these functions. To build the framework, we propose a novel heap analysis that models associative arrays and dynamic (prototype) objects. Next, we propose value analysis providing additional information necessary to resolve dynamic features. Finally, we propose a technique that automatically and generically combines value analysis and a heap analysis modeling associative arrays and prototype objects.

Keywords

Static analysis, dynamic languages, heap analysis, combining heap and value analysis

Anotace

Název *Statická analýza jazyků s dynamickými funkcemi*

Autor David Hauzar
`hauzar@d3s.mff.cuni.cz`
`(+420) 221 914 285`

Školitel Prof. František Plášil
`plasil@d3s.mff.cuni.cz`
`(+420) 221 914 266`

Katedra Katedra distribuovaných a spolehlivých systémů
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze
Malostranské nám. 25, 118 00 Praha, ČR

Abstrakt

Dynamické funkce programovacích jazyků, jako je dynamický typový systém, dynamické volání funkcí, dynamické vykonávání kódu a dynamické datové struktury, poskytují flexibilitu, která urychluje vývoj. Tyto funkce ale snižují množství informací, které jsou kontrolovány v době kompilace. To má za následek nižší výkon a větší chybovost programů. Tento problém je možné vyřešit pomocí technik statické analýzy. Dynamické funkce bohužel pro tyto techniky představují překážku a zásadně omezují jejich přesnost, spolehlivost a výkonnost. Abychom tento problém pomohli vyřešit, navrhujeme framework pro statickou analýzu, který automaticky řeší dynamické funkce, a tím umožňuje definovat přesné a spolehlivé statické analýzy podobně jako v případě, kdy program dynamické funkce neobsahuje. Aby bylo takový framework možné vytvořit, navrhujeme novou techniku heap analýzy, která modeluje asociativní pole a (prototypové) objekty. Dále navrhujeme analýzu hodnot proměnných, která zjišťuje další informace potřebné pro vypořádání se s dynamickými funkcemi. Nakonec navrhujeme techniku, která umožňuje automaticky a genericky kombinovat analýzu hodnot proměnných s heap analýzou.

Klíčová slova

Statická analýza, dynamické jazyky, heap analýza, kombinace heap analýzy a analýzy hodnot proměnných

Contents

1 Introduction **13**
- 1.1 Running Example . . . 14
- 1.2 Problem Statement . . . 16
- 1.3 Research Goal and Objective . . . 17
- 1.4 Contributions and Publications . . . 17
- 1.5 Structure of the Thesis . . . 18
- 1.6 Note on Conventions . . . 19

2 State of the Art **21**
- 2.1 Static Analysis of Dynamic Languages . . . 21
  - 2.1.1 Static Security Analysis . . . 21
  - 2.1.2 Type Analysis . . . 23
  - 2.1.3 Heap Analysis . . . 24
  - 2.1.4 Reducing Dynamic Information . . . 25
  - 2.1.5 Code Optimization . . . 26
- 2.2 Combining Heap and Value Analyses . . . 27

3 Goals Revisited **31**

4 Heap Analysis **33**
- 4.1 Motivation and Overview . . . 33
  - 4.1.1 Variables, Arrays, and Objects . . . 33
  - 4.1.2 Dynamic Accesses . . . 35
  - 4.1.3 Explicit Aliasing . . . 35
  - 4.1.4 Comparison to other languages . . . 36
  - 4.1.5 Overview of the Approach . . . 36
- 4.2 Formalization . . . 38
  - 4.2.1 Analysis State Space . . . 38
  - 4.2.2 Data-flow equations . . . 39
  - 4.2.3 Access Paths . . . 41
  - 4.2.4 Read Accesses . . . 41
  - 4.2.5 Write Accesses . . . 43
  - 4.2.6 Merge . . . 49
  - 4.2.7 Termination and Soundness . . . 52

4.3 Summary of Chapter 4 52

5 Framework to Static Analysis of Dynamic Languages 55
5.1 Motivation 56
5.2 Overview and Architecture 57
5.3 Intermediate Representation 58
5.4 Building IR 61
5.5 Analysis Domain 63
5.5.1 Declaration Analysis 63
5.5.2 Heap Analysis 64
5.5.3 Value Analysis 66
5.6 Lattice Order and Meet 67
5.7 Join and Widening 67
5.8 Transfer Functions 68
5.9 Summary Heap Identifiers 70
5.10 Summary of Chapter 5 71

6 Implementation 77
6.1 Analysis Framework 77
6.2 Eclipse-based Tool 79
6.2.1 AST-level Functionality 79
6.2.2 Static Analysis Functionality 81

7 Experimental Results 85
7.1 Scalability of Heap Analysis 85
7.2 Case Studies 87
7.2.1 Benchmark Application 87
7.2.2 Email Client 90
7.3 Summary of Chapter 7 95

8 Conclusion and Future Work 97
8.1 Open Issues and Future Work 99

References 102

1 Introduction

In recent years, there has been a rapid growth in popularity of dynamically typed languages [50] such as JavaScript, PHP, Perl, Python, and Ruby.

By employing runtime information, they typically provide dynamic features such as virtual and dynamic method calls, dynamic includes and code execution, duck typing, and built-in dynamic data structures. On one hand, dynamic languages provide flexibility for accelerating of development, and, on the other, they make programs more error-prone and less efficient since the information checked at compile time is reduced [60]. This is a significant problem, since a high level of security and performance of many applications written in dynamic languages can be of particular importance—consider, e.g., web applications typically developed in these languages.

Static program analysis gathers information about programs independently of their inputs. Usual answers that could static analysis provide are related to possible types and values of variables, the information whether variables can be influenced by the input, the information whether a variable is always assigned before it is used, and the information about heap locations to which a variable can point. This information can be used, e.g., for error detection, security analysis, program debugging, code optimization, and code refactoring. Thus, static program analysis can make programs written in dynamic languages both less error-prone and more efficient eliminating their major disadvantages.

Unfortunately, dynamic features pose major challenges to static analysis. To resolve these features, the end-user analysis (e.g. taint analysis) needs to be combined with other analyses. For instance, in case of dynamic type system, types of variables are completely unspecified and any interprocedural static analysis needs to be combined with type analysis to determine targets of method calls. Moreover, method calls and include statements can be dynamic in the sense that the name of the method to be called or the file to be included is specified by expression computed at run-time. That is, the resulting analysis must track values of variables and evaluate these expressions. In dynamic languages, all these data can be manipulated using

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. Employing heap analysis, modeling these data structures is essential for the precision of resulting analysis. Importantly, the value and the heap analyses must interplay. To resolve dynamic accesses to data structures, the heap analysis needs value analysis to evaluate value expressions and the value analysis must track values not only over variables, but also over array indices and object properties.

1.1 Running Example

In this section, a PHP code snippet in Fig. 1.1 is used to illustrate some of dynamic features that are challenging for static analysis.

At lines (1)–(9) classes for processing the output are defined. They can either log the output or show the output to the user. At lines (13)–(16) the application mode is set based on the value of `DEBUG` either to `log`—the application will log the output—or to `show`—the application will show the output to the user. At lines (17)–(20) the skin is set based on user input. At line (21), the array `$users` is initialized with the address of administrator. This value is not taken from any source of sensitive information and can be directly shown to the user. Note that the update at line (11) is correct even if the variable `$users` is uninitialized. In PHP, if a non-existing index is updated, it is automatically created. Moreover, if the update involves next dimension, the index is initialized with an empty array and another index is created in the next dimension. Next, at lines (23)–(24) information about the user name and user address is assigned to the array `$users`. At lines (25)–(26) data are processed to the output. Finally, at lines (27)–(36), function `logAdmin` is defined and used to log information about administrator.

The code uses the following dynamic features:

Dynamic function and method calls. Dynamic (indirect) function or method call is a call where a name of the function or method is specified by an expression. Consequently, to determine the target of a call, the analysis needs to track values of variables.

An example of dynamic method call is at line (25). The method call is specified with the variable `$mode`. Depending on the value of the variable, either method `log()` or method `show()` is called.

Duck typing. Duck typing defines the semantics of method calls in dynamically-typed languages. Because variables have not declared types (classes), to call a method, a variable is not enforced to be of a class that defines the method or inherits the method definition. Instead, at the time of the method call, the method is searched in the object to which the variable is pointing. If the object contains the method, it is called, otherwise the call results in a runtime error. Consequently, from the perspective of static analysis, duck typing is similar to virtual method calls—the analysis must compute information about objects to which the variable can point. However, to support prototype-based programming where methods can be dynamically added to objects, it is not sufficient to abstract objects by their types.

For the examples of duck typing, see lines (25) and (26). The variable `$t` can contain instances of classes `Templ1` and `Templ2` having the class `Templ` as common ancestor. The class `Templ` does not define the method `show()` that can be called here, thus in static language, the method could not be called. However, with respect to duck typing, the code is correct—both instances contain the method.

Dynamic data structures. Dynamic languages usually contain built-in support for dynamic variables, associative arrays, and prototype objects. Variables, array indices, and object properties need not be declared. If a specified index exists in an array, it is overwritten; if not, it is created. The same holds for object properties. Arrays as well as objects can have arbitrary depth.

Moreover, for some dynamic languages such as PHP and Perl, updates automatically create empty arrays and objects if also further dimensions are updated. For example, at line (11), the update automatically creates an array in a variable `$users` and creates another array in the index `$users['admin']`. Unfortunately, this makes it impossible to decompose updates of such structures. Splitting the update at line (11) into updates `$t = $users['admin']` and `$t['addr'] = get_..._db()` results in different semantics. While the update at line (11) creates an index containing an array in `$users['admin']` in the case it does not exist, the read access in `$t = $users['admin']` returns `null` and the subsequent update `$t['addr'] = get_..._db()` fails.

Note that in some other dynamic languages such as JavaScript and Python, updates do not automatically create arrays and objects and they can therefore be decomposed. However, even in these languages, the aforementioned semantics can be emulated using reflection. That is, the update at line (11) can be emulated in JavaScript in the following way:

```
if (typeof users['admin'] == 'undefined') {
```

```
    users['admin'] = [];
}
users['admin']['addr'] = 'admin';
```

Dynamic accesses to data structures. In dynamic languages, variables, indices of arrays, and properties of objects can be accessed with arbitrary expressions. Update can thus involve more than one element and can be even statically unknown. Since variables, arrays indices, and object properties that are updated and do not exist are created, the set of variables, array indices and object properties is not evident from the code. As an example, at line (23) the `$users` array with an index determined by the value of variable `$id` value, which is statically unknown, is assigned. Next, the update at line (24) is also statically unknown and may or may not influence the access at lines (26).

Dynamic name binding. The names of functions, classes, and constant are bound to concrete definitions during runtime. Consider the code at lines (27)–(36). If `DEBUG` is true, function `logAdmin` is defined per the first declaration and both address and name of the administrator is logged. Otherwise, the second definition is used and only administrator name is logged.

1.2 Problem Statement

Dynamic features provide flexibility that accelerates the development, in particular, the development of web applications. According to [53], most of the web applications are written in languages where such features are explicitly present. Web applications written in other languages often simulate dynamic features, e.g., using reflection, casting, and hash tables. Unfortunately, dynamic features make it impossible to use mature static analysis techniques. While there has been a lot of research concerning precise and scalable modeling of dynamic features, important features are still covered insufficiently, which limits the precision and scalability of existing techniques. The common source of imprecision is the modeling of dynamic data structures, such as associative arrays and prototype objects, which are used often, in particular in web applications. Next, there are no means for automatically resolving dynamic features and existing static analyses must deal with these features ad hoc. from scratch and in its own way. Thus, they became overly complex or imprecise. Consequently, there is a lack of specialized analyses for

dynamic languages, e.g., analyses for error detection, code optimization, and code refactoring.

1.3 Research Goal and Objective

The goal of the thesis is to make it possible to specify precise static analyses of languages with dynamic features in a simple way. In particular, the goals are (1) design precise heap analysis modeling data structures that are common in dynamic languages, (2) design techniques that will allow automatic resolving of dynamic features and thus allow defining static analyses independently of these features.

1.4 Contributions and Publications

Our first motivation for static analysis of dynamic languages stems from aiming at detecting security vulnerabilities in web applications via static taint analysis. In [23], we noticed that a precise value analysis together with heap analysis modeling associative arrays and objects is necessary for resolving dynamic constructs and crucial for the precision and soundness of a technique. To reduce the number of false-positive warnings, we proposed a technique of path-sensitive validation of security vulnerabilities. When prototyping static analyzer for PHP, we further elaborated these techniques (subsequently published in [24]).

In [26], we presented a novel heap analysis modeling associative arrays and prototype objects and backed it with a full formalization. It tackles the following challenges in static analysis of associative arrays: (1) Indices are not declared—if an updated index exists, it is overwritten, otherwise it is created. (2) Indices can be accessed using arbitrary expressions, which can yield even statically unknown values. Consequently, the set of indices employed for an array is not evident from the code. (3) Specifically for multidimensional arrays, updates cannot be decomposed. The reason is that updates create indices if they do not exist and initialize them with empty arrays if also further dimensions are updated; on contrary, read accesses do not.

To precisely resolve dynamic features, a value analysis modeling all of the primitive types, native operators, and implicit conversions needs to be defined. A fundamental challenge here lies in defining interplay between the value and heap analyses modeling associative arrays, prototype objects, and accesses to these data structures with arbitrary expressions. In this thesis,

we defined such interplay in a generic way, so that the heap analysis can be combined with arbitrary value analysis (including taint analysis) and each of them can be defined independently.

As a proof-of-the-concept, we designed a static analysis framework for PHP applications presented in [25].

Publications

[23] David Hauzar and Jan Kofron. Hunting bugs inside web applications. Technical report, Department of Informatics, Karlsruhe Institute of Technology (presented in FoVeOOS '11), 2011

[24] David Hauzar and Jan Kofron. On security analysis of php web applications. In *COMPSACW '12: Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 577–582, Washington, DC, USA, 2012. IEEE Computer Society

[26] David Hauzar, Jan Kofroň, and Pavel Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS '14: Proceedings of the 3rd International Workshop on Engineering Safety and Security Systems*, Electronic Proceedings in Theoretical Computer Science, pages 56–70. Open Publishing Association, 2014

[25] David Hauzar and Jan Kofron. WeVerca: Web verification for php. In *SEFM '14: Proceedings of the 12th International Conference on Software Engineering and Formal Methods*, Lecture Notes in Computer Science, Berlin, Heidelberg, 2014. Springer-Verlag

[44] Viliam Simko, David Hauzar, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Verifying temporal properties of use-cases in natural language. In *FACS '11: Proceedings of the 8th International Symposium on Formal Aspects of Component Software*, Lecture Notes in Computer Science, pages 350–367, Berlin, Heidelberg, 2011. Springer-Verlag

[45] Viliam Simko, David Hauzar, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Formal verification of annotated textual use-cases. In *Computer Journal (submitted)*, 2014 (current status: minor revision)

1.5 Structure of the Thesis

The thesis is structured in the following way: Chapter 2 provides an overview of existing approaches to static analysis of languages with dynamic features

and approaches to combining heap and value analyses. Chapter 4 focuses on heap analysis modeling dynamic data structures. Chapter 5 presents a framework for static analysis of dynamic languages that allows defining static analyses independently of dynamic features. Chapter 6 presents project WeVerca, which was developed in scope of this thesis. It consists of a static analysis framework for PHP and a tool for analysis of PHP web applications. Chapter 7 evaluates the scalability of the heap analysis and presents two case-studies evaluating the scalability and precision of the implemented tool. Finally, Chapter 8 concludes the thesis and proposes direction for future research.

1.6 Note on Conventions

The text of this work is partially based on the aforementioned publications. To distinguish a text included verbatim from the publications, corresponding paragraphs are marked with a vertical bar on the right or left side of the text. Where appropriate, the original text was slightly modified (in a way not changing the meaning) to make the thesis coherent and easy to read.

Here is an example of how paragraphs are marked to indicate a verbatim copy. It means that from this point, the text appeared in the paper [X]. [X]

```
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'] = 'admin';
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($users['admin']['addr']);
27 if (DEBUG) {
28   function logAdmin($templ, $users) {
29     $t->log($users['admin']['addr']);
30     $t->log($users['admin']['name']);
31   }
32 else
33   function logAdmin($templ, $users) {
34     $t->log($users['admin']['name']);
35   }
36 logAdmin($t, $users);
```

Figure 1.1: Running example

2 State of the Art

In this chapter, we describe state-of-the-art of static analysis of dynamic languages. Since we are aiming at designing heap analysis modeling dynamic data structures and we want to allow defining static analyses independently of modeling these data structures, we also examine the work on combining heap and value analyses.

2.1 Static Analysis of Dynamic Languages

Static analysis of applications developed in dynamic languages is a very hot research topic. This section presents its main application areas. The section provides a summary of the most influential approaches focusing on the way how these approaches deal with dynamic features.

2.1.1 Static Security Analysis

Dynamic languages are widely used in the development of web applications. Since these applications often store sensitive data and offer a wide spectrum of possible attacks to malicious users, security of these applications is of the primary importance. This is the reason why the work on static analysis of dynamic languages emerged from demands of security analysis and has drawn a rich body of techniques. Most of these techniques perform *static taint analysis*. Static taint analysis tries to find critical commands (*sinks*), which use data that can be manipulated by a potential attacker (*tainted* data).

The pioneering work in this area is the work of Huang et al. [27]. They developed a static analysis for PHP applications in WebSSARI tool and perform static taint analysis to identify vulnerabilities. Xie [57] discusses the limitations of their approach, in particular that it is intraprocedural and it

does not model dynamic features such as dynamic arrays, objects, dynamic variables, and dynamic includes.

The approach of Xie et al. [57] uses interprocedural analysis to find SQL injection vulnerabilities in PHP applications. To model sanitization process, they perform taint analysis. Sanitization can occur via calls to specified sanitization functions, casting to safe types, and a regular expression match. The analysis models automatic conversions of particular scalar types, uninitialized variables, dynamic accesses to associative arrays, and include statements. However, it leaves important parts of PHP unmodeled. In particular, it does not model multi-dimensional associative arrays, references, object oriented features of PHP, and it ignores recursive function calls.

Wasserman et al. [54, 55] use grammar-based string analysis following Minamide [39] to find a set of possible string values of a given variable at a given program point and gain this information to detect SQL injections. However, the employed static analysis has an incomplete support for references, does not track type conversions and omits modeling of associative arrays.

Pixy [31, 32] performs taint analysis of PHP programs and provides information about the flow of tainted data using dependence graphs [4]. Pixy performs 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, omitting type inference, and limited support for handling file inclusion and aliasing. Alias analysis introduced in Pixy incorrectly models aliasing when associative arrays and objects are involved.

Balzarotti et al. [4] extended Pixy to perform the analysis of the sanitization process. Their analysis is performed after Pixy computes control flow and relies on dependence graphs provided by Pixy. They represent values of variables at concrete program points using finite state automata tracking what parts of strings are not sanitized and perform string analysis through language-based replacement. The main limitation of their string analysis is that they approximate variables updated by loop as arbitrary strings. To filter-out false positive warnings, for each sink discovered by static analysis, they use dependence graphs provided by Pixy to generate code representing all possible sanitization operations manipulating tainted data along the path between the source of sensitive data and this sink. Finally, they exercise this code with a set of inputs simulating attacks. If a malicious input is not reduced to non-malicious values, the input is reported as a concrete example that violates the security of the application.

Yu et al. [58] also use dependence graphs computed by Pixy to perform

the analysis of the sanitization process. They developed an automata-based string analysis that enables to prove that an application is free from attack patterns specified as regular expressions. To tackle the problem of handling variables updated in loops, they incorporate the widening operator [5] and thus in these cases obtain better approximations than [4].

Andromeda static taint analyzer [51] 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 information which is then not available. To reduce this problem, Andromeda uses F4F [47], which reduces the amount of information that are not known statically.

Livshits et. al. [36] 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 the data-flow graph and thus on the precision of static analysis.

2.1.2 Type Analysis

Dynamic languages do not allow declaring types of variables and a single variable can contain values of different types depending on the context. As type information is necessary for computing control-flow and establishing the correctness of the program, there have been developed several type analyses meant as a basis for further program analyses and also for checking type properties of programs.

Phantm [34] 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 [30] is a JavaScript static program analysis infrastructure that in-

fers 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 the static analysis by constructing control-flow on-the-fly during the analysis. To capture information about dynamically allocated data, it uses recency abstraction [2]. Since TAJS models JavaScript semantics precisely, it has been successfully used to enable additional analyses. In [17, 18], TAJS program analysis infrastructure is used to build a tool for refactoring of JavaScript and in [29] 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 significant loss of precision in case of some other dynamic languages.

2.1.3 Heap Analysis

Heap analysis attempts to statically determine the structure of the heap. In the context of dynamic languages, which do not allow direct pointer manipulation, heap analysis usually computes the set of objects and arrays to which a variable, an object field, and an array index may point. Dynamic languages pose fundamental challenges here—they allow creating new array indices and object fields at runtime and accessing both arrays and objects via arbitrary expressions. Moreover, since updating non-existing array index or object field creates the index or the field and reading the index or the field returns undefined value, updates to associative arrays and objects cannot be decomposed. However, since information about heap structure is essential for soundness and precision of static analyses when objects are considered, there were developed several points-to analyses meant as a basis for further program analyses.

Sridharan et. al. [48] present static flow-insensitive points-to analysis for JavaScript. They model objects in JavaScript using associative arrays that can be accessed by arbitrary expressions. 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 being the variable. Thus, the correlation between the update and store is preserved. However, important limitation of their technique is that it assumes that updates to arrays and objects can be decomposed to updates of depth one.

Jang [28] presents flow-insensitive points-to analysis for JavaScript. It models variables, arrays, and objects using associative arrays. Limitations of their work include that it precisely models only assignments to constant indices—for all other assignments, a special unknown field is used. Moreover, the same as Sridharan [48], they assume that updates to associative arrays can be decomposed.

2.1.4 Reducing Dynamic Information

As 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 the static analysis.

F4F [47] 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.

Schafer et. al. [43] present a dynamic analysis for identifying variables and expressions that always have the same value at a given program point independently on the context. Such values can be used, e.g., to make constrained dynamic constructs static and thus enhance the scalability and precision of static analysis.

Phantm [34] reduces the number of information that static analysis must compute and possibly overapproximate by collecting this information at runtime. It first lets the application execute and collect this information and then invokes static analysis from a particular runtime state. The authors reported significant improvement of analysis precision, in particular when dynamic and nested data structures are used.

Wei et. al. [56] reduce the number of statically-unknown information in static analysis of JavaScript by using a technique of blended static analysis [15]. This technique collects statically-unknown information at run-time and uses it during static analysis. To collect statically-unknown information, it executes a test suite and collects execution traces of an application. Then, the technique processes each execution trace as follows. First, it extracts

run-time information from the trace. This information consists of call graph of the trace, types of created objects, and dynamically generated code. Next, static analysis of the application with respect to information from the execution trace is performed. That is, information from the execution trace is used to resolve dynamic constructs. Finally, solutions from different execution traces are combined into a single solution for the application. Authors show that by this resolving dynamic constructs using runtime information, they can significantly increase the scalability and precision of static analysis. However, the approach is not sound and error coverage depends on the quality of the test suite.

2.1.5 Code Optimization

Dynamic languages provide flexibility at the cost of performance. As there are a lot of applications developed in dynamic languages that are computational demanding, e.g., web applications, which can serve a huge number of requests, there have been attempts for automatic code optimization of programs developed in dynamic languages.

Zhao et. al. [60] tackle the performance problem of interpreting PHP by static compilation. They noticed that much overhead is caused by runtime type checking. Since variables can contain values of different types during execution, they are kept in generic, boxed values. Since PHP native operations are mostly untyped and have different semantics for different types of operands, in order to perform appropriate action, the types of operands are checked at runtime. To tackle this problem, they perform static type inference. For symbols whose types can be inferred, specific inferred types are used, for which the runtime system contains fast implementations. For static type inference, they soundly approximate the dynamic constructs and the analysis can infer specific types only in simple cases. While even this led to a significant gain in performance (approximately $2\times$), the authors claim that the precision of type inference plays a central role in their compiler. The more types can they statically infer, the better the performance would be. Consequently, by modeling dynamic constructs, much better results could be achieved.

Biggar et al. [7] perform context sensitive, flow sensitive, interprocedural static analysis of PHP in order to gain information usable for code optimizations in their PHP compiler. They combine alias analysis, type inference and literal analysis, model arrays, PHP's variable-variables construct, objects, references, scalar operations, casts, and weak type conversions. Unfortunately, the authors provide only an informal description, and details of

their method are not clear.

2.2 Combining Heap and Value Analyses

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[19, 52]. This problem of combining heap and static value analysis is particularly relevant for static analysis of dynamic languages. As programs written in dynamic languages often manipulate associative arrays and prototype objects, heap analysis modeling these data structures is necessary to allow precise and sound data propagation. Since these data structures can be dynamically accessed with arbitrary expressions, to resolve such accesses, the heap analysis needs to be combined with value analysis modeling usually intricate value semantics of dynamic languages. Unfortunately, to our best knowledge, there is no technique that generically combines heap and value analyses for dynamic languages.

Gopan et. al. [21] studied the problem of designing summarizing abstract numeric domains from existing numeric domains. In addition to classic numeric domain, which tracks values on variables, summarizing abstract numeric domain tracks values on summary objects. Summary objects represent potentially unbounded collections of numeric objects, e.g., heap-allocated objects and array elements abstracted by a single abstract object. This also happens when combining heap and value analysis—value analysis must track values also on heap identifiers, which are usually summary objects. Gopan et. al. discovered the following problems of summarizing value domains: (1) value analysis cannot perform strong update [35] when assigning a value to a summary object—since the update affects only one concrete object represented by the summary object and we do not know which one, we must keep all values that the summary object has before the update and only add assigned value. (2) Value analysis cannot correlate summarized objects to non-summarized objects. E.g., assigning value of summary object means that after the assignment, one concrete object represented by the summary object is equal to the target object. Not that the all the concrete objects represented by the summary object are equal to the target object. That is, the summary object is not equal to the target object. They solved these problems and extended existing numerical domains to summarized domains in a generic way.

Clousot [16] 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. The value analysis then tracks values on variables and also on these heap identifiers. While the approach allows using arbitrary value analysis, it only allows using specific heap analysis, the heap analysis cannot use information from value analysis, and their technique is not sound.

McCloskey et. al. [38] allow to combine heap and numerical value analyses in a generic way. The heap analysis splits the heap into disjoint regions and value analysis tracks values on variables and heap identifiers corresponding to these regions. While each analysis can choose to represent its abstract elements however it desires, to define semantics of combined analysis, they require each analysis to provide first order logic predicates, which are shared among the analyses. This can make the specification of combined analyses laborious. Importantly, they assume that assignments do not affect separation of heap into regions—the set of individuals belonging to the region is never changed. That is, they allow only to change predicates that hold over the members of a class. This makes it impossible to precisely model, e.g., adding of new object fields and array indices using dynamic updates.

Chang and Leino [8] face the problem of subexpressions that are unknown to given abstract domains (e.g. heap accesses are unknown to value domain) by using congruence-closure abstract domain parametrized by these abstract domains—base domains. The congruence-closure abstract domain stores congruence-closed equivalence classes of terms. These equivalence classes are represented as variables, giving base domains an illusion that these terms are just variables. Congruence-closure domain consults its base domains during such updates—by evaluating parts of such updates, which are known to a particular base domain, the base domain supply the congruence-domain information, which can the congruence-domain use to unify more terms and thus gain more precision. While the main advantage of their technique is that it can work with arbitrary (slightly extended) value domains and since equivalence classes may be dissolved as the variables of the program change, it can be extended to model adding of new object fields and array indices using dynamic updates, the main limitation is it forces the heap analysis to be based on equalities.

Miné et. al. [40] 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 [20] 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 on variables and also on heap identifiers. The method is both generic—it allows to reuse existing analyses as black-boxes—and automatic—it does not require to provide any annotations specific to a particular heap and value analysis. 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 limits 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 the same 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`.

The technique of Ferrara [19] generically combines numeric value analysis and heap analysis overcoming the limitation of flow-independent naming scheme. To manage the mapping of concrete references to abstract locations, it introduces the concept of substitutions. Substitutions allow heap analysis to materialize abstract locations, i.e., to replace a single abstract location in the pre-state with more abstract locations in the post-state. Substitutions also allow the heap analysis to summarize abstract locations, i.e., to replace more abstract locations in the pre-state with a single abstract location in the post-state. Importantly, the substitutions are propagated to value analysis and the propagation is proven to be correct.

3

Goals Revisited

As it is apparent from the Section 2, modeling of associative arrays and prototype objects in heap analyses for dynamic languages is not sufficiently covered. Existing techniques either do not model these data structures at all [27, 54], model them incorrectly [31], use coarse over-approximations [60], model them in a limited way—they precisely model only the semantics of updates that does not automatically create empty arrays and objects (i.e., allows to decompose multi-dimensional updates) [57, 30, 48], model only accesses to constant indices and fields [28], and model statically-unknown read accesses, but omits statically-unknown updates [34]. As these data structures are ubiquitous, especially in web applications, which manipulate a lot of input, this constitutes a significant source of unsoundness and imprecision.

Next, to resolve dynamic constructs and thus allow any static analysis, it is necessary to perform complex value analysis tracking values of all primitive types, modeling dynamic type system, native operators, and implicit conversions.

Importantly, to gain precise and sound results, heap and value analyses cannot be performed separately, but need to interplay. For example, value analysis needs information from heap analysis in order to correctly propagate value information and heap analysis needs information from value analysis in order to resolve dynamic index and property accesses.

Since heap and value analyses have been studied mainly as orthogonal problems, it is often not clear how to combine them in a precise and scalable way. While techniques of combining heap and value analyses exist [20, 19, 38, 8, 40], it is not possible to apply them for dynamic languages, where object properties and array indices can be added at runtime and accessed using arbitrary expressions. Existing techniques for static analysis of dynamic languages thus combine heap and value analyses ad hoc, which makes defining these analyses complex and error-prone. Consequently, these techniques often combine heap and value analyses inappropriately or omit some important analysis aspect thus sacrificing soundness, precision, and scalability.

To summarize, defining even simple static analysis for a dynamic language (e.g., taint static analysis), requires additionally to define the following things: (1) heap analysis, (2) complex value analysis tracking values of all primitive types of the language, and (3) the interplay of all analyses. This makes a huge barrier preventing static analysis to be more used in the context of dynamic languages.

In order to tackle these problems, the thesis aims at the following goals:

- **G1** *Design heap analysis for dynamic languages.*

  Design a heap analysis modeling built-in data structures used in dynamic languages. These include multi-dimensional associative arrays and objects. In particular, the goal is to model constant, non-constant, and statically unknown updates and read accesses to associative arrays and objects. Moreover, the heap analysis will precisely model the semantics of multi-dimensional updates to associative arrays and objects that automatically creates empty arrays and objects if also further dimensions are updated and thus does not allow decomposing the updates. This goal is reflected in Chapter 4.

- **G2** *Generically define interplay of heap and value analyses for dynamic languages.*

  Design a technique that will allow automatic combining of various heap and value analyses for dynamic languages. In particular, the technique will allow value analysis to track values not only on variables, but also on heap identifiers—array indices and object fields—and it will propagate changes done by the heap analysis to value analysis using standard operations of the value analysis domain. These changes include creating new heap identifiers both during the join operation and the assignment and updating heap identifiers during the assignment. This goal is reflected in Chapter 5.

- **G3** *Design framework for static analysis of dynamic languages.*

  Design a framework that will allow defining static analyses for dynamic languages (e.g., static taint analysis) independently of computing control flow and accessing data structures. In particular, this includes defining how control flow and accesses to data structures are resolved using information from value analysis. As a proof of the concept, static taint analysis will be defined. This goal is reflected in Chapter 5.

4 Heap Analysis

In dynamic languages, data are manipulated using built-in 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. Consequently, while tracking values is necessary for resolving dynamic features such as virtual and dynamic method calls and dynamic includes, in dynamic languages it cannot be done precisely and soundly without interplay with heap analysis modeling these data structures.

In this chapter we present our approach to these challenges. Our contribution includes: (1) creating heap analysis of associative arrays that can have arbitrary depth and can be accessed using variables and array indices containing even statically unknown values, (2) precise modeling of the semantics of multi-dimensional updates that does not allow decomposing the updates, and (3) modeling explicit aliases between variables, array indices, and object properties.

[26]

4.1 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; the main reason for this choice has been its worldwide usage making it the number one among the web-app languages. For the illustration of our concepts, we use the code in Fig. 4.1 as a running example.

4.1.1 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

```
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 the 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; // $alias[1] can still be aliased
     with $alias2
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 4.1: Associative arrays-like data structures.

line 7 in Fig. 4.1, 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 `$arr[1]`, but its copy. The second reason is that while updates create indices if they do not exist and initialize them with empty arrays when also further dimensions are updated, read accesses do not; while the update at line 7 creates an index `$arr[1]` in the case it does not exist and initialize it with empty array and then creates further index `$arr[1][2]` in this array, 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.

4.1.2 Dynamic Accesses

In dynamic languages, variables, indices of arrays, and properties of objects can be accessed with arbitrary expressions. At line 4 in Fig. 4.1 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 properties 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`.

4.1.3 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 the 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.

4.1.4 Comparison to other languages

We have chosen PHP as the language we would use in the rest of the paper for demonstration of our approach. Nonetheless, it is worth mentioning that many other languages, especially those connected with the development of web applications, provide built-in associative arrays-like data structures with similar semantics.

For example, the same way as PHP, Perl, JavaScript, Python, and Ruby provide associative arrays and objects with indices and properties that can be added at runtime. Moreover, Perl provides the same semantics of multi-dimensional updates as PHP. This semantics does not allow to decompose the updates without the loss of precision. It initializes new array indices and object properties with empty arrays and objects when also further dimensions are updated (see, e.g., the assignment at line (7) of Fig. 4.1). Note that in other dynamic languages, this semantics can be emulated using reflection. That is, reflection can be used to check whether updated index exists and if not, the index can be created and explicitly initialized with empty array or empty object.

Moreover, to ease the development, 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.

[26]

4.1.5 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 representation 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. Note that specifically for updates, this is necessary to take into account that the updates cannot be decomposed. 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 be assigned to the new index using previous 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`). 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]`.

4.2 Formalization

We formalized our data-flow analysis using data-flow equations for the forward data-flow analysis [41]. 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.

4.2.1 Analysis State Space

Tab. 4.1 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

[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.

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

Table 4.1: 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 4.2: Special variables and values.

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. 4.2 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. 4.3, there are several helper functions (projections) defined; we use them in the subsequent definitions.

4.2.2 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 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 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)	$values(x, V \in \mathcal{P}(Var))$	$= \{values(x, v), v \in V\}$
(4)	$values_{undef}(x, V \in \mathcal{P}(Var))$	$= values(x, V)$ if $V \neq \emptyset$
		$= \{undefined\}$ if $V = \emptyset$
(5)	$values(x, undefVar)$	$= \{undefined\}$
(6)	$indexOf(x)$	$= i$
(7)	$indices(x, V \in \mathcal{P}(Val))$	$= \{iv, \exists_{v \in V} \exists_{n \in Val}((iv, n), v) \in i\}$
(8)	$indices(x, V \in \mathcal{P}(Var), I \in \mathcal{P}(Val))$	
		$= \{iv, \exists_{v \in V} \exists_{ind \in I}((iv, ind), v) \in i\}$
(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)	$aliases_{must/may}(x, V \in \mathcal{P}(Val))$	$= \{(a, v), a \in aliases_{must/may}(x, v),$ $v \in V\}$
(12)	$aliases(x, v \in Val)$	$= aliases_{may}(x, v) \cup aliases_{must}(x, v)$

Table 4.3: List of helper functions and projections.

4.2.3 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.

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: $\$a[\$b]$–$[r(s)][a][[r(s)][b]]$, $\$\$a$–$[r(s)][[r(s)][a]]$, $\$a[\$b[\$c]][2]$–$[r(s)][a][[r(s)][b][[r(s)][c]]][2]$.

4.2.4 Read Accesses

Tab. 4.4 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

#	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 $AP = [][AP_1]...[AP_n]$ $\wedge Vars_n(x, v, AP) = \emptyset$ (b) $= Vars_0(x, v, AP)$ if $AP = []$ (c) $= Vars_n(x, v, AP)$ if $AP = [][AP_1]...[AP_n]$ $\wedge Vars_n(x, v, AP) \neq \emptyset$ (d)
(16) (17)	$Vars_0(x, v, AP) = \{v\}$ $\forall_{i \in 1,...,n}$: $Vars_i(x, v, AP) = indices(x, Vars_{i-1}(x, v, AP))$ 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)	$indices_r(x, V \in \mathcal{P}(Var), I \in \mathcal{P}(Val)) = indices(x, V, I) \cup$ (a) $\bigcup_{v \in V, ind \in I} \{u, ((u, \bullet), v) \in indexOf(x)$ (b) $\wedge \nexists_{w \in Vars} ((w, ind), v) \in indexOf(x)\}$

Table 4.4: Definition of read accesses.

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 4.1. 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—$[][arr][\bullet]$.

4.2.5 Write Accesses

Tab. 4.6 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. 4.7 defines $KILL$ and GEN sets, which contains data that are removed and added to these variables. Tab. 4.5 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. 4.6 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.

[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 omit it from the formalization to make the presentation of our approach more clear.

Example 5: An example of a weak update is the update at line 4 in Fig. 4.1—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. 4.1. 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 4.5: 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 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.

#	**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)	$must \ = must_n \land may \ = may_n$
(23)	$must' = must'_n \land may' = may'_n$
(24)	$indices_w(V \in \mathcal{P}(Var), I \in \mathcal{P}(Val))$ $= indices(IN, V, I) \cup$ $\bigcup_{v \in V, ind \in I} \{ defindex(v, ind),$ $\nexists_{w \in Var}((w, ind), v) \notin indexOf(IN \cup GEN')\}$
(25)	$defindex(v \in Var, ind \in Val)$ $= \{i, ((u, \bullet), v) \in indexOf(IN \cup GEN') \land$ $i = createindex(GEN', v, ind) \land$ $deepcopy((IN \cup GEN', u), (GEN', i))\}$
(26)	$must_0 = \{r(IN)\} \land may_0 = \emptyset$
(27)	$\forall_{j \in 1,2,...,n}$ $((\lvert I_j \rvert = 1 \land I_j \neq \{*\}) \land ($ (a) $must_j \ = aliases_{must}(indices_w(must_{j-1}, I_j)) \land$ $may_j \ = aliases(IN'_k, indices_w(may_{j-1}, I_j)) \cup$ $aliases_{may}(IN, indices_w(must_{j-1}, I_j)))) \lor$ $((\lvert I_j \rvert > 1 \land * \notin I_j) \land ($ (b) $must_j \ = \emptyset \land$ $may_j \ = aliases(IN'_k, indices_w(may_{j-1} \cup$ $must_{j-1}, I_j)))) \lor$ $(* \in I_j \land ($ (c) $must_j \ = \emptyset \land$ $may_j \ = aliases(IN'_k, indices(IN'_k, may_{j-1} \cup must_{j-1}))))$
(28)	$((\lvert I_n \rvert = 1 \land I_n \neq \{*\}) \land ($ $must'_n \ = indices^j_w(must_{j-1})) \land$ $may'_n \ = indices^j_w(may_{j-1}))) \lor$ $((\lvert I_j \rvert > 1 \land * \notin I_j) \land ($ $must'_n \ = \emptyset \land$ $may'_n \ = indices^j_w(may_{j-1} \cup must_{j-1}))) \lor$ $(* \in I_n \land ($ $must'_n \ = \emptyset \land$ $may'_n \ = indices(IN'_k, may_{n-1} \cup must_{n-1})))$

Table 4.6: Definition of collecting variables for an update.

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 `unknown` 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. 4.1 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. 4.1. 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. 4.1. 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. 4.1, 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. 4.7 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 4.1. The *must* set is empty, the may set consists of variables $[][arr][1]$, $[][arr][2]$,

#	Expression	
	Assignment / 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 4.7: Definition of updates for assignment and alias statement and new object expression.

[][arr][•], [][$arr2$][1], [][$arr2$][2], [][$arr2$][3], and [][$arr2$][•]. 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]. 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., [][$arr2$][•] is a may-alias of [][$alias$] and [][$arr2$][•]. 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][•] and [][$alias$]. Thus, the subsequent read access using access path [][arr][3] would read also value 9.

4.2.6 Merge

Tab. 4.8 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. 4.1 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 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. 4.1 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

#	**Expression**
(36)	$mergeStates(OUT \in \mathcal{P}(\Sigma)) = \{IN, mergeVars((IN, r(IN)), \{(o, r(o)), o \in OUT\})\}$
(37)	$mergeVars(R \in \Sigma \times Var, M \in \mathcal{P}(\Sigma \times Var)):$ $APs = \bigcup_{(x,v_r)\in M}(accessPaths(x, v_r, [])) \cup \{[]\} \wedge$ (a) $ResAPs = extend(APs) \wedge$ (b) $\forall_{AP \in ResAPs}(mergeAP(R, M, AP))$ (c)
(38)	$mergeAP(R = (x_R \in \Sigma, v_R \in Var), M \in \mathcal{P}(\Sigma \times Var), AP):$ $resVar = createVar(R, AP) \wedge$ (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) $aliases_{may}(x_R, resVar)$ (e) $= \bigcup_{(x_m,v_m)\in \text{mergedVars}}\{aliases_{must}(x_m, v_m) \cup aliases_{may}(x_m, v_m)\} - aliases_{must}(x_R, resVar) \wedge$ $indices(x_R, resVar)$ (f) $= \bigcup_{(x_m,v_m)\in mergedVars} \{createVar((x_R, resVar), [][n]), n \in indicesNames(x_m, v_m)\}$
(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)	$newAP([][v_1]...[v_n], l \in Int, V \in \mathcal{P}(Val))$ $= \{[][u_1]...[u_n], \forall_{i=1,...,n \wedge i \neq l}(u_i = v_i) \wedge u_l \in V\}$ 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 4.8: Definition of merge.

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. 4.1. 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.

4.2.7 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 [34] or implicitly by using, e.g., allocation-site abstraction [1, 9] 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. 4.6, Tab. 4.7, and Tab. 4.8 are monotonic so the fixpoint computation terminates.

Our heap analysis 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.

4.3 Summary of Chapter 4

In this chapter, we described dynamic accesses to associative arrays and prototype objects, which are common in dynamic languages, but make it hard

to apply static analysis. In particular, we described the semantics of non-decomposable multidimensional updates to associative arrays and prototype objects. This semantics creates new array indices and object properties that are accessed during the update if they do not exist and initializes them with empty arrays and objects when also further dimensions are updated.

As a solution to this problem, we proposed heap analysis modeling dynamic accesses. Since prototype objects can be modeled using associative arrays, we described the analysis in terms of associative arrays.

We defined how read and write data accesses are performed and how analysis states are joined. To resolve data accesses and compute the shape of the heap precisely, the heap analysis tracks also values of variables and array indices. Similar to [48, 30, 28], we model updates of statically unknown indices by employing special index called unknown field. The main contribution we made was that unlike existing techniques, we took into account the semantics non-decomposable multidimensional updates to associative arrays. We model such updates soundly and precisely even if statically-unknown data from the input are used to specify targets of the updates.

Our technique has two important limitations. First, we assume expressions that contain just index-access (property-access) operators and simple abstract domain for tracking values, which is defined together with heap analysis. To the contrary to this, in dynamic languages, values of many primitive types can be used and expressions can involve other operators and implicit conversions. This requires complex abstract value domain and complex transfer functions for this domain. Combining such value analysis with heap analysis ad hoc is complex and error-prone.

Second, we assume that the control-flow graph (CFG) of the application is given as an input of the analysis. While this assumption is realistic for static languages, for dynamic languages with (dynamic) method calls and dynamic includes, CFG must be computed during the analysis.

To overcome these limitations, Chapter 5 presents a static analysis framework for dynamic languages. The framework builds CFG during the analysis. To gain all necessary information, it uses value analysis tracking values of all PHP primitive types, modeling dynamic type system, native operators, native functions, and implicit conversions. Importantly, the framework defines the interplay between heap and value analysis and thus allows defining these analyses independently of each other. As the interplay is generic, it not only simplifies definitions of these particular analyses, but allows independent definition of additional heap and value analyses (e.g., taint analysis).

5

Framework to Static Analysis of Dynamic Languages

To analyze programs precisely and soundly, the static analysis (e.g., taint 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. The dependence is also the other way around—since array indices and object properties can be accessed with arbitrary expressions, the heap analysis needs value analysis to evaluate these expressions. This makes any end-user static analysis overly complex.

In this chapter we present modular static analysis framework for languages with dynamic features. The framework automatically resolves dynamic features and makes it possible to define static analyses without taking these features explicitly into account.

In particular, the contributions we made are the following:

- We present the architecture of the framework and how dynamic features are automatically resolved.
- We define value analysis tracking values of all primitive data types of PHP. Its main contribution is the design of a lattice structure that fits with the purpose of providing enough information for automatically resolving dynamic constructs.
- We define the interplay between a value analysis and a heap analysis that models associative arrays and (prototype) objects. The interplay allows defining both analyses independently and it also allows combining different value and heap analyses. Here the main challenge is to take dynamic index and property accesses into account—indices and

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

Table 5.1: Propagation of tainted data.

properties are created when they are accessed for the first time and accesses can be made with arbitrary expressions, yielding even statically unknown values.

5.1 Motivation

[25] As a motivational 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., 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, queries a database, etc. is referred to as *sink*. Data at a given program point are *tainted* if they can pass from a source to this program point. A tainted data are *sanitized* if they are processed by a sanitization routine (e.g., `htmlspecialchars` in PHP) to remove potential malicious parts of it. Program is *vulnerable* if it contains a sink that uses data that are tainted and not sanitized.

Static taint analysis can be performed by computing the propagation of tainted data and then checking whether tainted data can reach a sink. The propagation of tainted data computed by forward data-flow analysis is shown in Tab. 5.1[1]. The analysis is specified by giving the lattice of data-flow facts, the initial values of variables, the transfer function, and the join operator.

[25] Consider now the code in Fig. 1.1. The code contains two vulnerabilities. At lines (25) and (26) the method `show` of `Templ1` can be called, its parameter `$msg` can be tainted and the parameter goes to the sink. Taint analysis defined using our framework uses just the information in Tab 5.1

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

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). Moreover, as the framework automatically reads the data from and updates the data to associative arrays and objects, at line (24), the tainted data are automatically propagated to index `$users['admin']['addr']` defined at line (11). Consequently, the access of this index at line (26) reads tainted data.

5.2 Overview and Architecture

[25]

The architecture of the framework is shown in Fig. 5.1. 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 is stored in the data representation. IR defines the order of instructions' execution and has function calls, method calls, includes, and exceptions already resolved. In the second phase, end-user analyses of the constructed IR are performed.

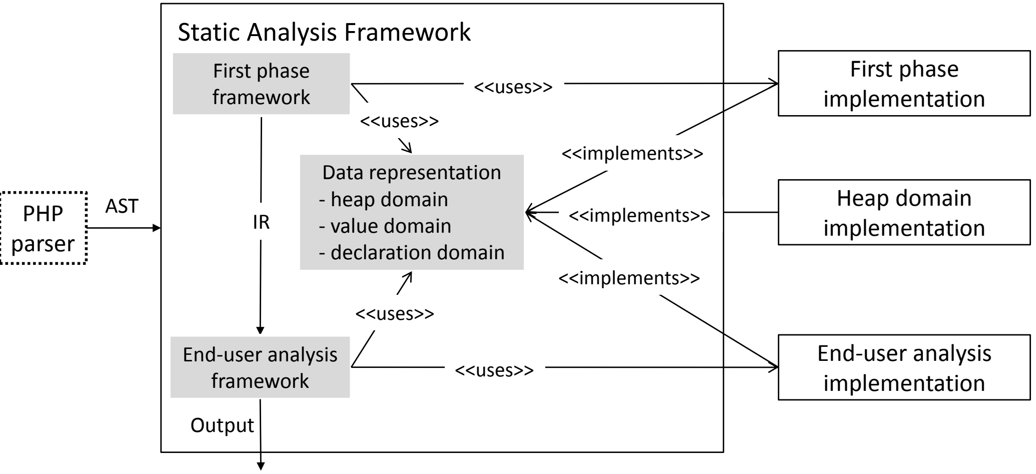


Figure 5.1: Architecture of the framework.

Data representation stores analysis states—data of heap, value, and declaration analysis—and allows accessing analysis states. In particular, it allows reading values from data structures, writing values to data structures, and modifying shape of data structures. Next, it performs join and widening of analysis states and defines their partial order. Importantly, data representation defines the interplay of heap, value, and declaration analyses allowing each analysis to define these operations independently of other analyses.

The implementation of the first phase must provide information necessary for computing control flow of the program and accessing 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, the implementation of the first phase must compute targets of throw statements, include statements, function and method calls, and it must define context sensitivity.

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 be specified using an arbitrary value domain. This is possible because (1) control flow is already computed, (2) the shape of the heap is computed and dynamic data accesses are resolved (i.e., value expressions specifying data accesses are evaluated). That is, all information that data representation needs to discover which variables, array indices, and object properties are accessed is available. (3) Data representation combines heap, value, and declaration analyses automatically. That is, to perform operations with analysis states, it uses standard operations of combined analyses.

5.3 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 that program instructions can be executed. *Value edges* connect nodes that use values with nodes that represent these values.

Each node has associated an analysis state stored in data representation. The state is modified by transfer function defined for the node and the re-

sulting 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 the 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 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 intermediate representation corresponding to the statement `$$a=b($c)`. The statement assigns the value computed by function `b` to a variable with name given by the value of variable `$a`. The resulting intermediate representation is depicted in Fig. 5.2. 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—`$`). Next, the latter node is connected using a value edge with the node representing possible names of the assigned variable (the node `$a`).

The nodes can be of different types. In the following, we denote value nodes by adding superscript V. Next, nodes are connected with the value nodes that are their arguments using value edges:

variable$^V[n^V]$: represents a variable—stores the information necessary for accessing the variable in data representation. N^V is the value node that represents a 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 given object. O^V is the value node storing a 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 variable, object property, or array index.

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 feasible. If the condition is unfeasible, the flow is not propagated to the 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 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. That is, e.g., the function to be called in a single call point can be determined by an expression that can yield more concrete values and thus more functions can be called in a single call point.

return$[e^V]$: represents a return from a function. The value node 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, *path* can represent more concrete values.

eval$[c^V]$: evaluates a code specified by value node c^V.

native-method$[]$: represents execution of native method or native function.

extension$[f, a]$: follows call^V, construct^V, include, and eval^V nodes. During the analysis, the control flow of these nodes is extended using a single extension node for each function, method, and constructor that may be called, script that may be included, and code that may evaluated. Extension node is followed by an initial node of the graph that corresponds, e.g., to the body of the function. Parameter f is a node that is extended and parameter a is a list of value nodes representing parameters of a call. These parameters are used to initialize function and constructor calls—for example to bind actual parameters to formal parameters.

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

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

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

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

5.4 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 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 from these nodes is set to be empty.

The control flow of caller nodes is extended during the static analysis. When processing a caller node, the analysis framework provides the first phase implementation all information computed by the analysis so far relevant to determine the control flow. Using this information, the first phase implementation finds appropriate function and method definitions and 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

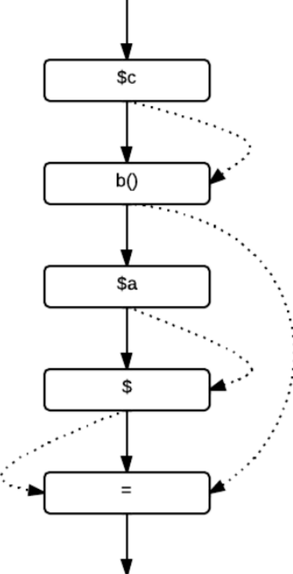


Figure 5.2: Intermediate representation of the statement `$$a=b($c)`. Solid edges are flow edges, dashed edges are value edges.

shared between multiple caller nodes. This way, the first phase implementation can control context sensitivity.

Finally, the control flow of the caller nodes is extended with these IRs. 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 extension node binds actual parameters to formal parameters for function, method, and constructor calls, extension-sink joins states of final nodes of all the IRs that extend the corresponding caller node.

Example 2: Fig. 5.3-A shows IR when it is initially built and the caller node is not extended. Fig. 5.3-B shows the intermediate representation 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 possible types. Fig. 5.3-C shows the case when a single IR is shared by multiple callers.

5.5 Analysis Domain

The states of our abstract domain have a form of State $= H \times V \times F$ where H is a state of the heap analysis, V is a state of the value analysis, and F is a state of the declaration analysis. The heap analysis tracks the shape of the heap and approximates concrete 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 on both.

5.5.1 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 F is a set of class, function, and constant declarations and lattice operators of the analysis are $\langle F, \subseteq, \cup, \cap \rangle$.

Example 3: Consider the following PHP code:

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

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

5.5.2 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 even statically unknown values. If a specified variable, index, or object property exists, it is overwritten; if not, it is created. Note that when a variable, index, or property is created, there could be statically unknown assignments that could update it and this new variable, index, or property should be initialized with corresponding values.

To be able to capture this semantics, heap analysis approximates arrays, objects, array indices, object fields, and even variables[2] with heap identifiers and it can manage these heap identifiers. In particular it can materialize identifiers from existing identifiers (i.e., create new identifiers that are initialized using existing identifiers) both during assignment and join operation. These changes are propagated to value analysis.

To enable materialization of heap identifiers that have not been certainly updated by any statically-unknown assignment, the value component always contains the heap identifier `??` representing undefined value. Such identifiers are materialized from this heap identifier.

Example 4: Fig. 5.4 shows the heap and value component of the state after the update at line 23 in Fig. 1.1. We use adopted heap analysis developed in Section 4 and 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 an array `Root` representing a symbol table. The array contains three heap identifiers (`id`, `users`, and `?`), which represent program variables (`$id` and `$users`) and statically unknown variables. For the heap identifier `id`, the value analysis tracks the value `AnyString`, while the heap identifier `users` is present only in the heap domain and points to an array. The heap identifier `users-admin` represents the index admin of the array, while the heap identifier `users-?` represents statically unknown indices of the array. Both heap identifiers point to arrays representing next dimensions. 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.

[2] Variables are treated as indices of associative array representing symbol table.

Heap identifiers are accessed using the function read $\in AE \mapsto \mathcal{P}(\text{HId})$ provided by the heap component. The function returns a set of heap identifiers identified by given *access expression*. Access expression is obtained from nodes of type variable^V, property-use^V, and index-use^V. In case of variable^V, access expression is the set of values, in case of property-use^V, and index-use^V, it is a sequence of sets of values. Every set from the sequence contains values that can be used to access corresponding dimension of an array or corresponding object in object reference chain. That is, access expressions can represent multi-dimensional updates. This is necessary to model semantics of updates that does not allow decomposing them.

Example 5: Consider reading an index `$users[10]['name']` from the state depicted in Fig. 5.4. The access expression for the index is $\{users\}$ $\{10\}\{'name'\}$. The function read returns the heap identifier `users-?-name`. This heap identifier is then used to read values `UndefString` and `AnyString`, which correspond to undefined value and statically-unknown value.

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

We assume that the heap analysis is provided with lattice operators $\langle H, \sqsubseteq_h, \sqcup_h, \sqcap_h \rangle$. The operator $\sqsubseteq_h$ specifies the partial order, $\sqcup_h$ is join operator, and $\sqcap_h$ is meet operator. The semantics of heap analysis is given by a transfer function $[\![\bullet]\!]_h \in H \mapsto H$.

Moreover, we assume that the heap analysis provides a function joinToValue $\in H \times H \mapsto \mathcal{P}(\text{HId} \times \text{HId}) \times \mathcal{P}(\text{HId} \times \text{HId})$ that for each joined state returns pairs of heap identifiers, where the first identifier in each pair is the identifier that is materialized from the second identifier in the pair (i.e., the first identifier should be inserted to the joined state and it should be initialized with values of the first identifier).

Finally, we assume that the heap analysis provides a function assignToValue $\in H \times \text{APE} \mapsto \mathcal{P}(\text{HId} \times \text{HId}) \times \mathcal{P}(\text{HId}) \times \mathcal{P}(\text{HId})$ that returns the heap identifiers that are materialized by the assignment statement, the heap identifiers that certainly must be updated, and the heap identifiers that may be updated.

5.5.3 Value Analysis

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

Second phase. Value domain for the second phase tracks information over heap identifiers and it is provided with lattice operators $\langle H, \sqsubseteq_{v_2}, \sqcup_{v_2}, \sqcap_{v_2} \rangle$, transfer function $[\![\bullet]\!]_{v_2} \in V_2 \mapsto V_2$, and widening operator ∇_{v_2}. It can be fully specified by the user of the framework.

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

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

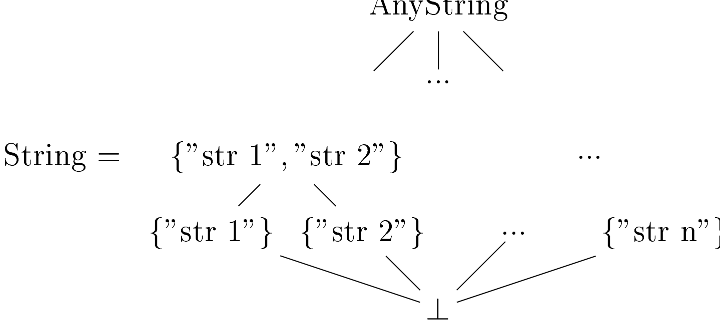
Since PHP has dynamic type system—variables, array indices, and object properties do not have declared types, and they can store values of different types depending on context—, Value_1 can store values of all primitive types.

For numeric component Num, we use interval domain. Other components are described by the following lattices:

Undef = undef | ⊥

Null = null | ⊥

Bool = AnyBool / true, false / ⊥



String component is represented by a lattice of sets that contain strings. The size of sets is limited by a constant, thus the lattice is finite.

Example 6: The abstract value $(\bot, \bot, \text{AnyBool}, \bot, \bot)$ represents a concrete value that is of type boolean, the abstract value (undef, $\bot, \bot$, true, {"foo", "bar"}) represents a concrete values that is either `undefined`, the boolean `true`, the string `"foo"`, or the string `"bar"`.

For all components of Value_1 except of Num, we define widening operator ∇ to be equal to the join operator $\sqcup$.

5.6 Lattice Order and Meet

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

$$(h_1, v_1, f_1) \sqsubseteq_{\text{state}} (h_2, v_2, f_2) \iff h_1 \sqsubseteq_h h_2 \wedge v_1 \sqsubseteq_v v_2 \wedge f_1 \subseteq f_2$$
$$(h_1, v_1, f_1) \sqcap_{\text{state}} (h_2, v_2, f_2) = (h_1 \sqcap_h h_2, v_1 \sqcap_v v_2, f_1 \cap f_2)$$

5.7 Join and Widening

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

$$(h_1, v_1, f_1) \sqcup_{\text{state}} (h_2, v_2, f_2) = (h_1 \sqcup_h h_2, v_1' \sqcup_v v_2', f_1 \cup f_2)$$
$$(h_1, v_1) \nabla_{\text{state}} (h_2, v_2) = (h_1 \sqcup_h h_2, v_1' \nabla_v v_2', f_1 \cup f_2)$$
$$(n_1, n_2) = \text{joinToValue}(h_1, h_2)$$
$$v_1' = \bigsqcup_{(t,s) \in n_1} [\![t = s]\!]_v(v_1)$$
$$v_2' = \bigsqcup_{(t,s) \in n_2} [\![t = s]\!]_v(v_2)$$

Declaration and heap parts of input states are joined independently on other parts. To perform the join of value parts, the heap component provides the value component information about heap identifiers that are materialized in each joined state. This information is provided via the function joinToValue. For each joined state, the function returns a set of pairs (t, s) where t is the heap identifier that is materialized from the heap identifier

s. That is, the heap identifier t should be added to the value component of the state and initialized with values of the heap identifier s, which is already present in the value component. Note that s contains values of statically unknown assignments that could update the new identifier t.

After the information from the function joinToValue is provided, both value parts are updated with new identifiers using assignment transfer function for the value domain. Finally, the updated value parts are joined using the join operator for the value domain.

Example 7: Fig. 5.5 shows joining value and heap components of two states (v_1, h_1) and (v_2, h_2). For brevity we omit declaration components. Again, we use adopted heap domain from Section 4 for heap component and set domain for value component.

For the first state to be joined, heap identifiers `arr-1-3`, `arr-1-?`, and `arr-?-?` are materialized from the heap identifier `??` representing undefined heap identifier. That is, there were no statically-unknown assignments that could update these identifiers. These identifiers are thus added to the value component and initialized with `UndefString`. For the second state, heap identifiers `arr-1-?`, `arr-1-2`, and `arr-1-3` are added to the value component. Note that since there was statically-unknown assignment that could update the latter identifier, this identifier is materialized from the identifier `arr-?-3`. It is thus initialized with values `UndefString` and `second`.

Note, that the resulting value components v_1' and v_2' have the same set of heap identifiers. Finally, the join is performed component-wise.

5.8 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 makes allows getting access expression, which provides information necessary for accessing the value represented by the node. The access expression for the parameter l^V is l^V.AE, the access expression for the node r^V is r^V.AE.

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

$$\llbracket \text{assign}^V[l,r] \rrbracket_{\text{state}}(h,v,f) = (\llbracket l.\text{AE} = r.\text{AE} \rrbracket_h(h), v'', f)$$
$$(n, l_{must}, l_{may}) = \text{assignToValue}(h, l.\text{AE})$$
$$v' = \bigsqcup_{(t,s)\in n} \llbracket t = s \rrbracket_v(v)$$
$$v'' = \bigsqcup_{t\in l_{must}, s\in \text{read}(h, r.\text{AE})} \llbracket t = s \rrbracket_v(v') \sqcup \bigsqcup_{t\in l_{may}, s\in \text{read}(h, r.\text{AE})} v' \sqcup \llbracket t = s \rrbracket_v(v')$$

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 perform the transfer function for the value part of the state, the heap domain provides the value domain necessary information via the function assignToValue. This information consists of: (1) n—a set of pairs of heap identifiers representing heap identifiers that are defined by the assignment and their initial values, (2) heap identifiers that are certainly targets of the assignment, and (3) heap identifiers that may be targets of the assignment. The same way as in case of creating new heap identifiers during the join, each heap identifier t that is defined by the assignment is materialized from the heap identifier s that already was in the state before the assignment. The identifier s contains values from statically unknown assignments that could update the new identifier t.

After the information from the function assignToValue is provided, the value component is updated using its transfer function for the assignment. First, new heap identifiers are defined, second the heap identifiers representing targets of the assignments are updated. Note that the heap identifiers that only may be targets of the assignment are weakly updated. That is, since it is not certain whether these identifiers are updated by the assignment, after the assignment, they either can have the original values, or the new values. This effect can be approximated by $\lambda v.v \sqcup \llbracket t = s \rrbracket_v(v)$.

Example 8: Fig. 5.6 illustrates the transition function for the assignment at line 24 in Fig 1.1. 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 update the heap component of the state. During the update, the heap component materializes the heap identifier `users-?-addr`. This change is propagated to value component via the function assignToValue. Note that since there have not been any statically unknown assignments that could update this heap identifier, it is materialized from the identifier `??` representing undefined values. That is the identifier `users-?-addr` is added to the value component and initialized with `UndefString`.

Finally, the function assignToValue specifies that identifiers `users-?-addr` and `users-admin-addr` are weakly updated. Since the target of the assignment is not statically known, there are no heap identifiers to be strongly updated.

The definition of the transfer function for the node alias^V is analogous to the definition of the transfer function for assign^V. Again, the transfer function for the heap part is not influenced by the value part and the heap part provides value part information of the same form.

5.9 Summary Heap Identifiers

Value analyses are designed to track information on local variables, while we use value analyses to track information on heap identifiers that can represent many concrete heap locations—summary identifiers. Consider, e.g., 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 heap identifiers, it must take into account that they represent more heap locations.

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

```
5 $a = $users[$_GET[1]];
```

```
6 $b = $users[$_GET[2]];
7 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 $[\![a = users{-}?]\!]_v$ and the semantics of assignment at line (2) as $[\![b = users{-}?]\!]_v$. If v was relational domain the analysis would relate both identifiers `a` and `b` with the summary identifier `users-?` and incorrectly infer that the `if` branch can never be reached. This problem was studied by Gopan [21] et. al., who showed that it is wrong to correlate summarized objects with non-summarized objects and proposed the way how existing relational domains can be extended 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 implemented so far were also non-relational. To use relational value analyses, these analyses need to be extended to summary dimensions and the framework has to specify which heap identifiers are summary.

5.10 Summary of Chapter 5

Dynamic languages are challenging for static analysis: features such as dynamic type system, dynamic method calls, and dynamic includes imply that the control flow must be computed together with value analysis. Moreover, since data structures such as associative arrays and objects are frequently used, to gain sound and precise results, value analysis must be combined with heap analysis. This is, however, fundamentally complicated by ubiquitous use of dynamic index and property accesses. To define even simple but sound and precise static analysis (e.g., taint analysis), all these challenges need to be addressed. This makes a huge barrier preventing static analysis to be more used in the context of dynamic languages.

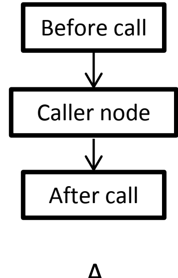
To solve this problem, we introduced a framework for static analysis of dynamic languages. The framework splits the analysis into two phases. While in the first phase, the framework computes control flow and information for accessing data structures, in the second phase, information from the first phase is used for end-user analyses.

The control flow is captured in the intermediate representation (IR). Since the control flow is not known before the analysis, the framework builds IR during the analysis. To provide the framework values necessary for computing control flow and values for resolving dynamic index and property

accesses, we defined a lattice structure for a first-phase value analysis. The lattice structure provides information about values of PHP primitive types that a heap identifier (i.e, a variable, an array index, and an object property) can store at a program point. Because of dynamic type system, a heap identifier can store values of more types at a single program point.

In order to allow heap and value analyses to be defined independently and to allow automatic combining of various heap analyses with arbitrary value analyses, we defined the interplay of value and heap analyses. To take into account dynamic index and property accesses, the heap analysis can materialize heap identifiers both during the join and assignment and the framework automatically propagates these changes to the value analysis. The interplay supports non-relational value analyses; however, it can be extended to also support relational value analyses.

Since end-user analyses use control flow computed in the first phase and the interplay allows value analyses to automatically access variables, array indices, and object properties, value end-user analyses can be defined without taking dynamic dynamic features explicitly into account.



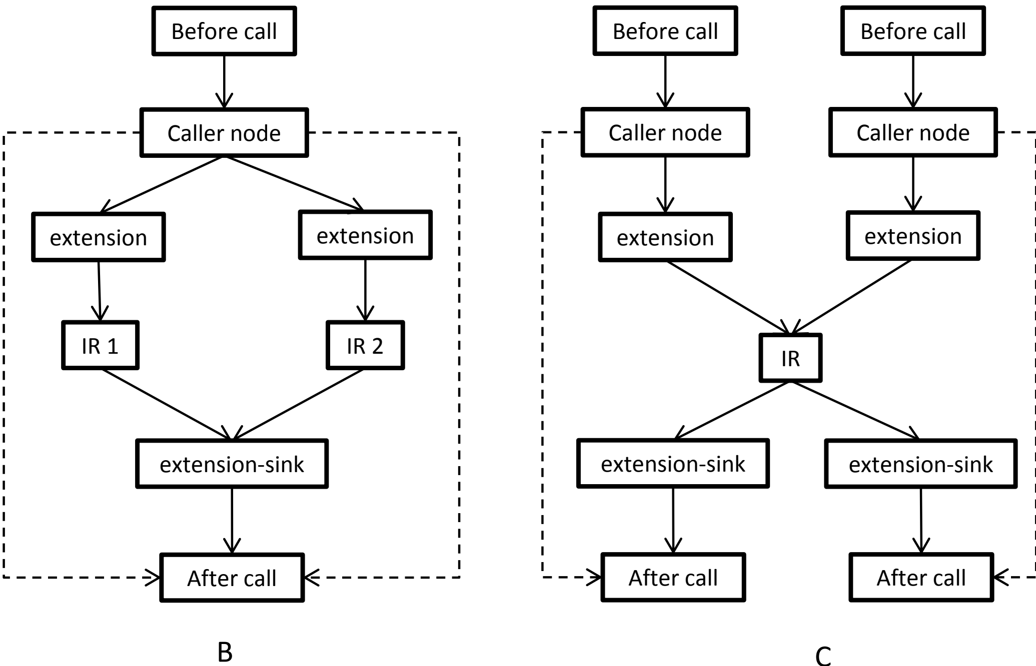
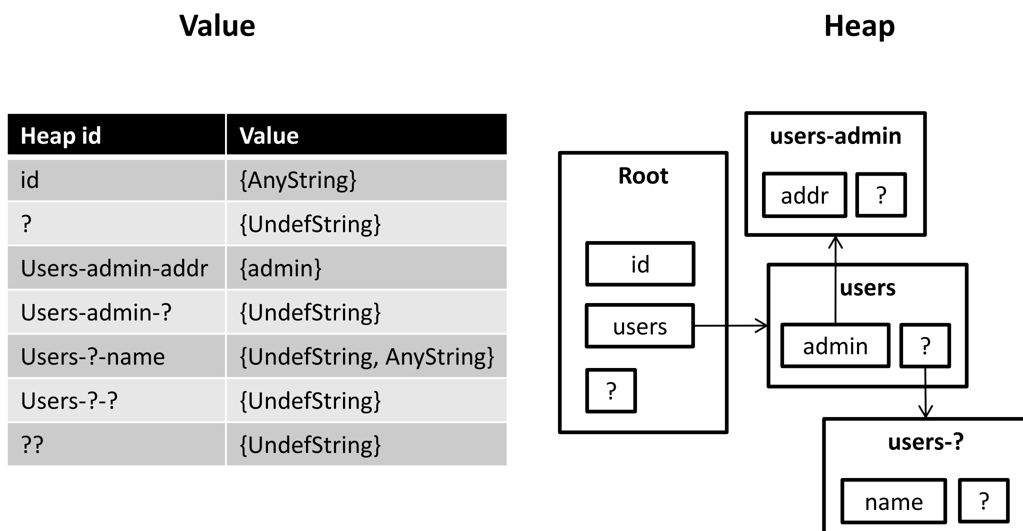


Figure 5.3: Building IR. Initial IR—the control flow of the caller node has not yet been extended (A). IR after processing the caller node during static analysis. The control flow of the caller node is extended with two IRs—IR 1 and IR 2 (B). Single IR shared between multiple caller nodes (C).



Heap id	Value
id	{AnyString}
?	{UndefString}
Users-admin-addr	{admin}
Users-admin-?	{UndefString}
Users-?-name	{UndefString, AnyString}
Users-?-?	{UndefString}
??	{UndefString}

Figure 5.4: The heap and string part of the value component of the state after the update at line 23 in Fig. 1.1.

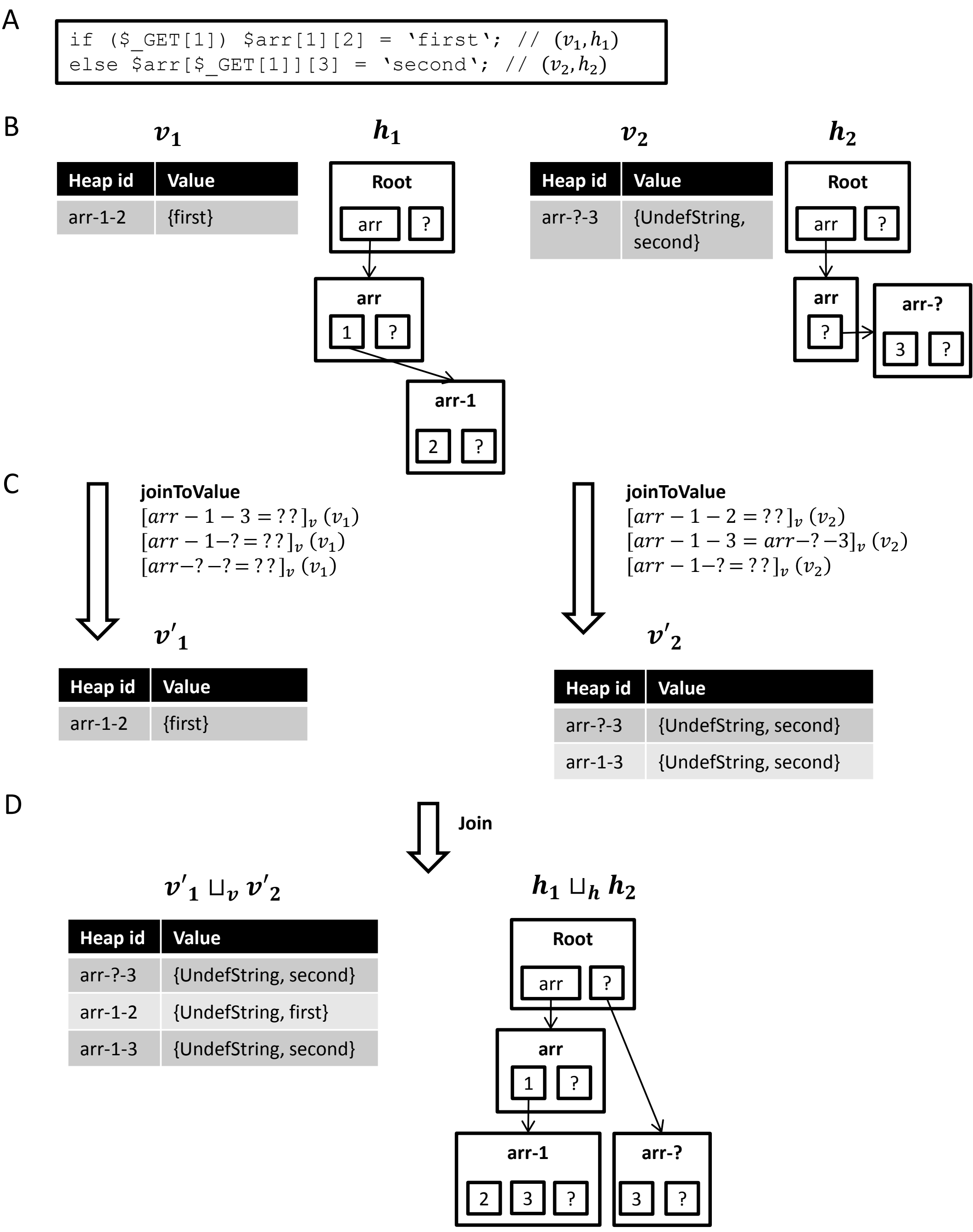


Figure 5.5: Joining value and heap components of two states (v_1, h_1) and (v_2, h_2). The corresponding code (A), value and heap components of joined states (B), adding new heap identifiers to value components of joined states (C), result of the join (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

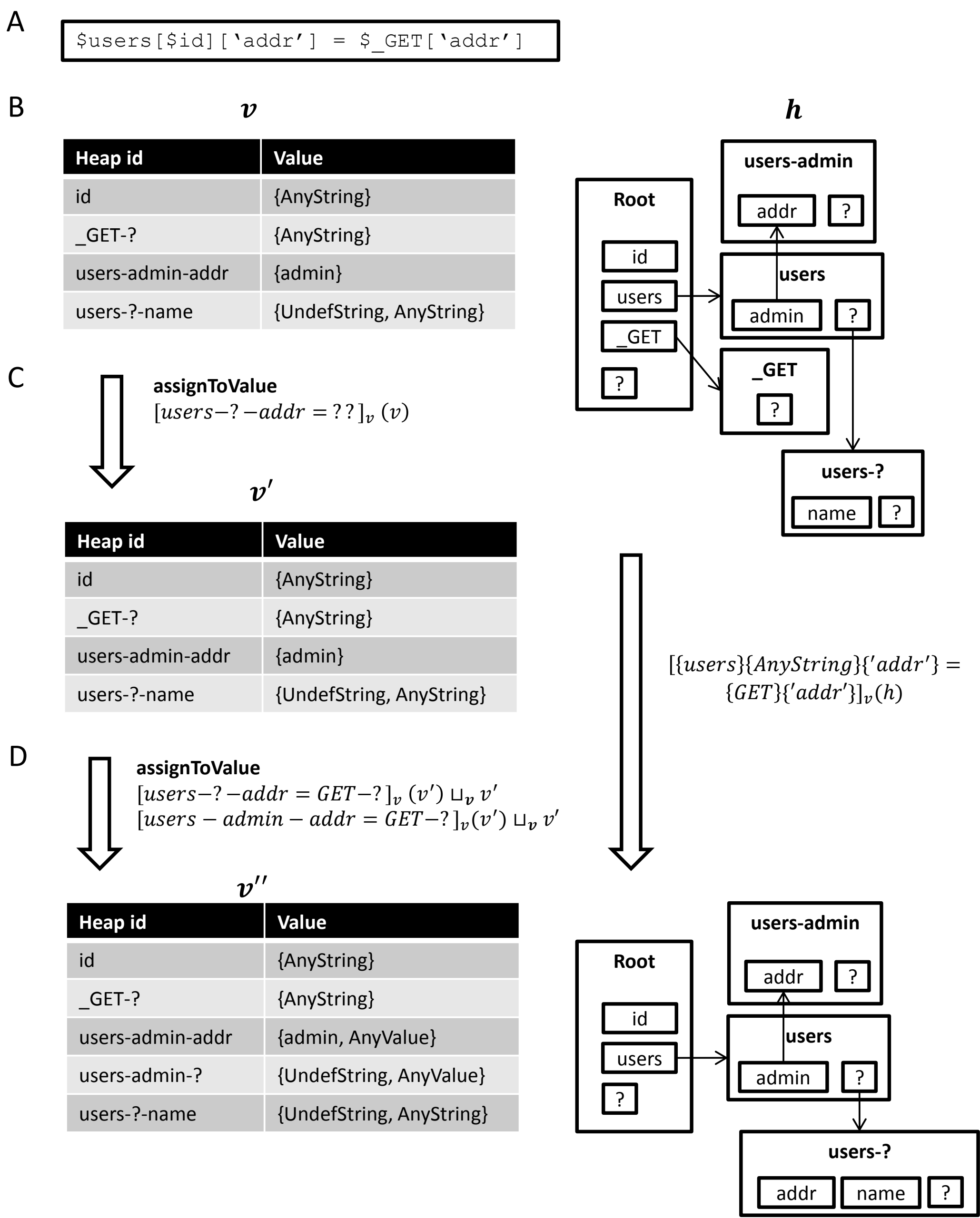


Figure 5.6: Transfer function for the assignment. The code of the assignment (A). The value and the heap component (v, h) of the state before the assignment (B). Adding the new identifier `users-?-addr` to the value component of the state (C). The value component v'' and the heap component of the state after the assignment (D). For the sake of space, the heap identifiers that have just value `UndefString` are not depicted in value components.

6 Implementation

As a proof-of-the-concept of our research, we created analysis framework for PHP applications [25] and we uses the framework to implement a tool for analysis of PHP applications integrated to Eclipse IDE. Both the framework and the tool are parts of WeVerca (WEb VERifiCAtion for PHP) project and are available at `http://d3s.mff.cuni.cz/projects/formal_methods/weverca/`.

6.1 Analysis Framework

The architecture of the framework is depicted in Fig. 6.1. To parse PHP sources and get abstract syntax tree (AST), the framework uses Phalanger [42, 6]. The rest of the framework is divided into two parts—AST-level analysis framework and static analysis framework.

AST-level framework traverses AST and provides API for implementing code metrics and AST-level checks.

Static analysis framework provides API for implementing static analyses. The architecture of static analysis framework follows the architecture described in Chapter 5. We implemented the following components:

- **Data Representation** stores analysis states (data of value, declaration, and heap analyses), performs join and widening of analysis states, compares the states, and provides API for manipulating and reading the states. Data representation also provides API allowing both first-phase and end-user analyses implementations define type of data tracked by the value analysis, define partial order on this data, and define join and widening operators applied to this data.

  WeVerca contains two implementations of heap analysis. The first implementation follows the semantics described in Chapter 4 supporting associative arrays as well as objects and accesses to these structures using even statically unknown values. The second implementation of

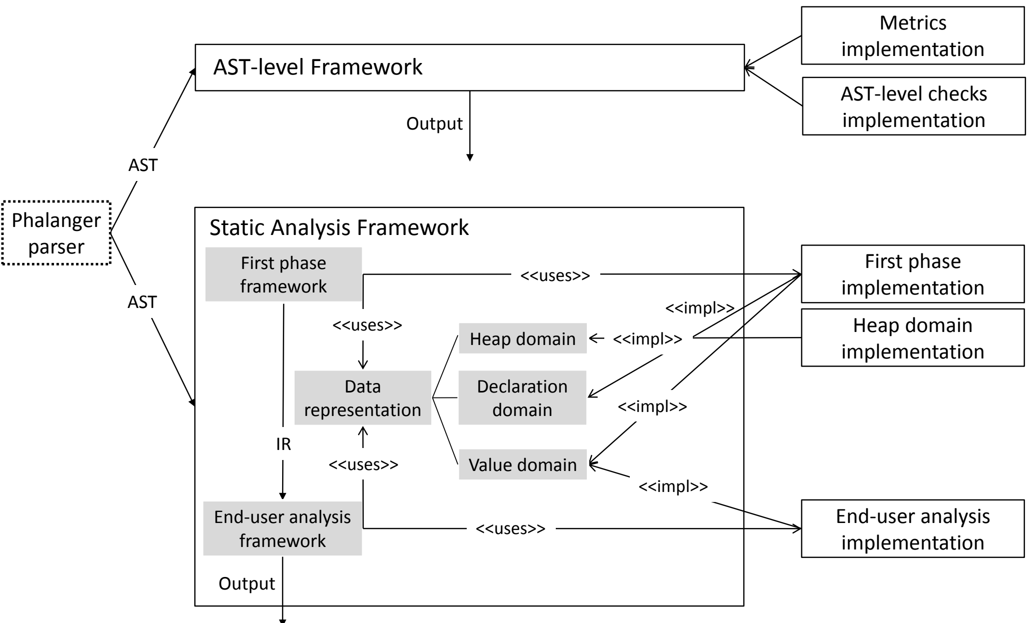


Figure 6.1: Architecture of WeVerca framework. Impl stands for implements.

heap analysis is designed with focus on scalability and does not precisely model accesses with statically unknown values and provides less precise modeling of explicit aliases.

- **First-phase framework** uses information provided by first-phase implementation and data representation to build intermediate representation (IR), controls evaluation of expressions, resolving dynamic accesses to data structures, and handling declarations. Next, it defines API for first-phase implementation. First-phase implementation must define value and declaration analyses providing first-phase framework information for computing control flow and accessing data structures and provide first-phase framework specification of context-sensitivity.

- **First-phase implementation**. WeVerca contains default implementation of declaration analysis tracking declarations of constants, functions, and classes and value analysis, which tracks values of all PHP primitive types, precisely models native operators, native functions, and implicit conversions. Default implementation of the first phase also specifies full context sensitivity.

- **End-user analysis framework** provides API allowing end-user analysis to perform actions at each IR node being processed during anal-

ysis. Note that while first-phase analysis must track values necessary for computing control flow and accesses to data structures, end-user analyses use control flow and accesses to data structures computed in the first phase.

Thus, to implement end-user analysis, it is just necessary to (1) specify values stored by the analysis, define partial order on these values, specify join and widening operators applied on these values and (2) use API provided by end-user analysis framework to specify how values are manipulated at IR nodes—to read and write values, use API provided by data representation.

- **Implemented end-user Analyses**. WeVerca contains implementation of taint analysis. In addition to information whether data are tainted, it tracks information necessary for computing all the potential sequences of assignments from the source of sensitive data to the sink. Next, since the routines for sanitization data differ depending on the sink and source (e.g., there are different routines for sanitization data for usage in SQL commands and sanitization data for sending it to the browser), it also distinguishes between various taint flags .

6.2 Eclipse-based Tool

Using the framework, we created a tool for analysis of PHP applications. The tool is implemented as a set of plug-ins for Eclipse IDE, which allows analyzing PHP applications in a convenient way. It provides functionality based on both AST-level analyses and static analyses.

6.2.1 AST-level Functionality

The tool uses AST-level analyses to provide the following functionality:

- **Code metrics**. The tool allows displaying the following code metrics:
  - **Number of lines**
  - **Number of sources**
  - **Maximum inheritance depth**
  - **Maximal depth of method overriding**
  - **Class coupling**

  - **Functions coupling**

- **Suspicious constructs**. The tool allows to display warnings about potentially dangerous constructs and functions:
  - **Database functions.**
  - **Session functions**. Read or write data that preserve across subsequent requests.
  - **Autoload**. Use of function `__autoload` or `spl_autoload_register`. These functions allow to load a type that has not been yet declared.
  - **Magic methods**. Methods that are called automatically when some operation with the enclosing object is performed (e.g., when a property of the object is read or written).
  - **Class presence**.
  - **Explicit aliasing**.
  - **Nested function declaration**. Functions and classes declared inside functions and methods. Locally declared function or class becomes global after the first call of enclosing function or method, however it cannot be redeclared. Since calling the function or method that declares another function or class more than once results in runtime error, we consider nested function declaration a bad practice.
  - **Use of super global variable**. Represent, e.g., an application input (`$_GET`, `$_POST`, `$_REQUEST`), session data (`$_SESSION`), and files uploaded to the current script (`$_FILES`).
  - **Dynamic dereference**. Specifying name of a variable using value of another variable, e.g., `$$a` is a variable whose name is given by the value of the variable `$a`.
  - **Dynamic call**. Specifying name of the function or the method to be called using value of variable.
  - **Dynamic include**. Specifying the file to be included using an expression.
  - **Eval**. Evaluating dynamically-generated code.
  - **Passing by reference at call site**. In PHP, parameter of a function or method can be specified to be passed by reference (i.e., the function can change its value) either at the callee site

(i.e., when declaring a function) or at the call site (i.e., when calling a function). The latter way is deprecated, as the function should specify its contract.

6.2.2 Static Analysis Functionality

The tool uses static analysis to gather information providing:

- **Unreachable code**.
- **Abstract call stack** for each program line representing all concrete call stacks for the program line that can happen at runtime.
- **Shape of the heap and values of variables, array indices, and object properties** for each program line and context.
- **Warnings for semantic problems**. For each semantic problem, the tool shows abstract call stack allowing to identify what is the cause of the problem. The errors that the tool detects are of the following categories:
  - **Security errors**. The tool uses taint analysis to identify security vulnerabilities detecting problems that allow cross-site scripting, SQL injection, and file include manipulation attacks.
  - **Call errors**. Errors that can occur during calls of functions, calls of methods, and file inclusions. These errors include calling function that is not declared, including file that does not exist, wrong number of arguments, calling inaccessible method, and wrong type of arguments for calls to native functions.
  - **Call resolving errors**. Errors indicating that the analyzer was not able to resolve a call or a file inclusion. These include an error indicating that the analyzer cannot compute names of functions to be called and that it cannot compute paths to files that should be included.
  - **Array, object, and class accesses errors**, e.g., accessing non-existing property, accessing inaccessible field, cannot use the operator `->` on non-object, class constant does not exist, cannot access parent class, and cannot index string with negative numbers.
  - **Class and interface declaration errors**, e.g., cannot re-declare class, cannot override function, and cannot redeclare interface method.

  - **Other errors**, e.g., division by zero, converting object to integer by arithmetic operation, and cannot instantiate abstract class.

- **Taint flows** for security errors, the tool allows visualizing flows of sensitive data from sources of sensitive information to critical commands. For each flow, the tool displays all assignments of sensitive data.

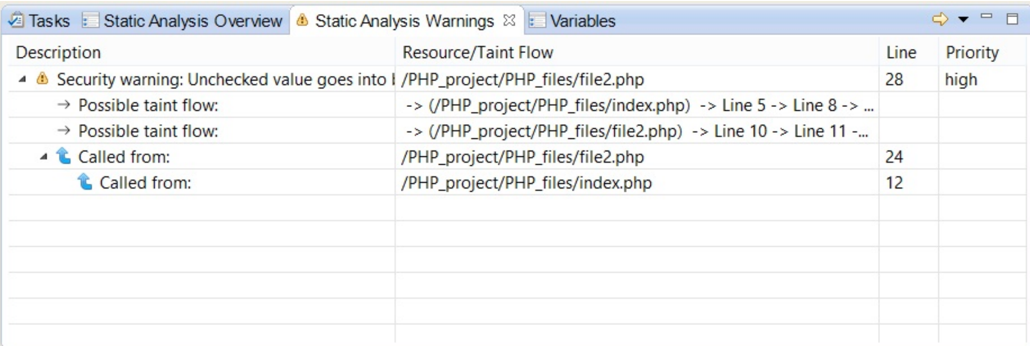


Figure 6.2: The screenshot from our Eclipse-based tool showing information about a security warning discovered by taint analysis. This includes abstract call stack and information about involved sink together with possible flows of tainted data to the sink.

Figure 6.3: The screenshot from our Eclipse-based tool showing visualization of a flow of tainted data from two sources (lines 4 and 5) to the sink (line 28). Tainted variables are shown in red.

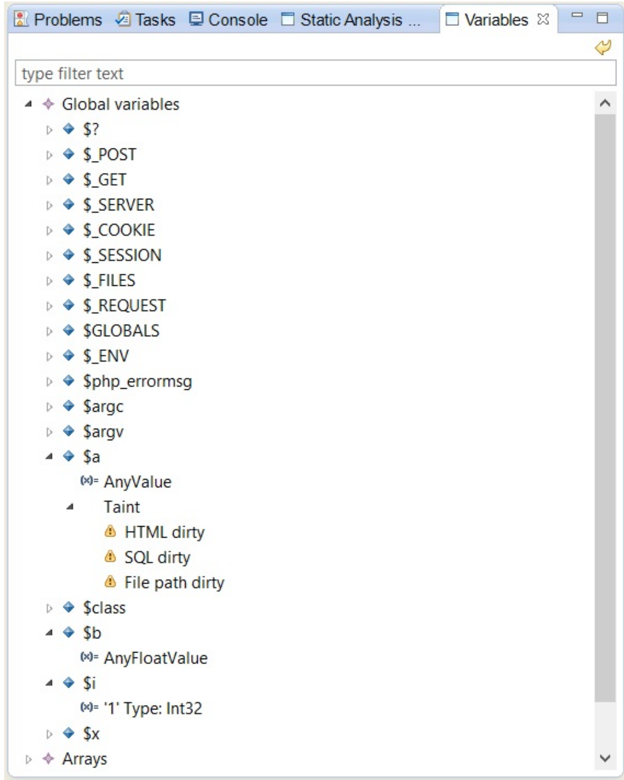


Figure 6.4: The screenshot from our Eclipse-based tool showing visualization of a shape of the heap and values of variables, array indices, and object properties computed by the analysis for a particular program point.

7 Experimental Results

This chapter presents experimental evaluation of our techniques using the proof-of-the-concept implementation WeVerca described in the previous chapter. In particular, we present an evaluation of the scalability of the heap analysis presented in Chapter 4 on synthetic PHP codes and two case studies conducted on real PHP codes.

7.1 Scalability of Heap Analysis

The novelty of heap analysis described in Chapter 4 is that it soundly and precisely models the semantics of non-decomposable multidimensional updates to associative arrays-like data structures even if statically-unknown data from the input are used to specify targets of such updates. Other analyses, such as [31, 60, 57, 30, 48] are more limited, e.g., they model the semantics in that the updates can be decomposed. The fundamental question we want to answer therefore is thus how our analysis scales.

[26] To evaluate our heap analysis we used the code `CODE_n` that was generated from the code fragment at lines (2)–(19) in Fig. 4.1 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 read and write 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 affected by the merge operation at each join point is constant. That is, each branch being merged modifies only constant number of variables. Since we want to also know how the merge operation of non-trivial states scales with respect to the number of variables affected by the merge operation, in addition to `CODE_n`, we generated the code `mCODE_n`. The code `mCODE_n` (Fig 7.1) is defined using `CODE_n` in a way that both branches merged by the top-level merge operation modify all variables defined in `CODE_n` and all the variables are thus affected by the merge operation.

Tab. 7.1 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

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

Figure 7.1: The code mCode_n used for the evaluation.

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 7.1: Number of nodes in control-flow graph, number of variables, and analysis times for generated codes.

number of variables defined in the analysis state of the program end point, and the running time of the analysis. From the results it follows that when the number of variables is small, the analysis is fast. Note that our testing codes are created so that complex data accesses are performed for all n—computation time of both write and read accesses depends only on the complexity of data access and does not depend on the number of variables in the state. That is, the results show that the only factor that impacts the analysis time significantly is the number of variables.

Tab. 7.2 shows the scalability of the merge operation with respect to number of variables affected by the merge. The results show that the dependence between the number of variables and merge time is about linear. Bad scalability of the heap analysis for large number of variables is caused by the

n	**Variables**	**Merge time (ms)**
1	107	3
2	211	6
3	419	12
4	835	27

Table 7.2: The scalability of the merge operation with respect to the number of variables affected by merging. For each n, the top-level merge operation in mCode_n was measured—all the variables defined in the mCode_n are affected by the merge operation.

fact that in our current implementation there is no sharing of data between the states. That is, all data stored in a state of a CFG node are copied to states of its successor nodes. Note that CFG nodes usually modify only small portion of analysis state and copying of unmodified data is superfluous. For large number of variables, such superfluous copying of data takes most analysis time and causes that the analysis does not scale for large number of variables.

We believe that by optimizing the implementation, the scalability of the analysis with respect to number of variables can be highly improved and our analysis can scale up to thousands of variables and tens of thousands nodes of CFG.

7.2 Case Studies

We used the tool presented in Section 6.2 implemented as a part of the WeVerca project, to analyze two PHP applications with a total of over 16,000 lines of PHP code. We analyzed these applications also with Pixy [31] and Phantm [33], the state-of-the-art tools for security analysis and error discovery in PHP applications, and compared the results.

7.2.1 Benchmark Application

Benchmark application comprises of a fragment of myBloggie weblog system[1]. The application contains known set of 13 problems, which allows us to assess error coverage and false positive rate for each analysis tool.

[25]

Tab. 7.1 shows the summary of results. WeVerca outperforms the other tools both in error coverage and number of false positives. Since WeVerca is much more precise than compared tools, we believe that slightly longer analysis time is a good pay off.

Identified Problems

WeVerca discovered all problems in the benchmark application. We now describe a selection of them.

1) Misplaced sanitization. For sanitizing user input, myBlogie frequently uses function `intval`, which converts its parameter to integer. WeVerca

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

	WeVerca	Pixy	Phantm
Warnings (#)	16	16	43
Error coverage (%)	**100**	69	23
False positive rate (%)	**23**	44	93
Time (s)	2.2	**0.6**	2.5

Table 7.3: Analysis of the benchmark application comprising of 648 lines of code.

found one case where this function misplaced—it is called after the assignment instead of before the assignment:

```
intval($comment_id = $_GET['comment_id']);
$sql = "... WHERE comment_id=$comment_id ...";
$result = mysql_query("... WHERE comment_id=
   $comment_id ...");
```

2) Missing initialization. In several places, a variable can be uninitialized and it is sent to the browser or used in database query. The simplified version of the corresponding code is:

```
if  (isset($_SERVER['HTTP_HOST'])) {
  $myUrl = "http://".$_SERVER['HTTP_HOST'].$pathweb;
}
elseif (isset($_SERVER['SERVER_NAME'])) {
  $myUrl = 'http://'.$_SERVER['SRV_NAME'].$pathweb;
}
echo $myURL;
```

If neither `$_SERVER['HTTP_HOST']` nor `$_SERVER['HTTP_NAME']` is set, the variable `$myURL`, which is sent to the browser, remains uninitialized.

False Alarms

WeVerca reported 3 false alarms. We now describe causes of these false alarms.

[25]

1) Modeling of function date. The first false alarm reported by WeVerca is caused by imprecise modeling of built-in function `date`. The simplified version of the corresponding code is:

```
$lang['January'] = 'January';
$lang['February'] = 'February';
```

```
...
$lang['December'] = 'December';

$month=gmdate('n', time() );
$year = intval($_GET['year']);
$monthIndx = date('F', mktime(0, 0, 0, $month, 1,
   $year));
echo $lang[$monthIndx];
```

WeVerca only models the return value of the function `data` by its type and deduced that the function can return any string value. However, while the first argument of the function is `"F"`, the function returns only strings corresponding to English names of months. The value returned by this function is used to access the index of an array id$lang, which is initialized with indices corresponding to English names of months, WeVerca incorrectly reports that an undefined index of the array can be accessed. Note that this false-alarm can be resolved just by modeling the built-in function `data` more precisely. [25]

2) Path-insensitivity. Two remaining false alarms are caused by path-insensitivity of the analysis. The simplified version of the corresponding code is: [25]

```
$post_id = $_GET['post_id'];

if ($mode == "viewid") {
  $post_id = intval($post_id);
}
...
if ($mode == "viewid") {
  $query = "... post_id='$post_id' ...";
  $result = mysql_query($query);
}
```

The sanitization and sink commands are guarded by the same condition; however, there is a joint point between these conditions that discards the effect of sanitization from the perspective of path-insensitive analysis. [25]

To filter out these false alarms, one can either use the technique of path-sensitive validation of alarms proposed in our paper [24] or some technique of path-sensitive static analysis [61, 12, 3, 46, 11, 49, 13]. The former technique is more efficient than the latter techniques and can be implemented quite easily using our framework. In general, it allows filtering less false positives; however, we believe, it could filter out most of false positives due to path-insensitivity in real-world web applications.

7.2.2 Email Client

For the second case study, we used a NOCC webmail client[2]. Tab. 7.4 shows summary of results for WeVerca analyser. To assess false positive rate, we inspected all reported warnings and determine whether the warning is a false alarm. Since we do not know the set of all security problems in NOCC, we were not able to assess error coverage.

[25] We do not show results for Pixy and Phantm. As Pixy supports only version 4 of PHP and NOCC uses object-oriented features introduced in version 5, Pixy is not able to analyze NOCC. Phantm was able to analyze NOCC in two minutes and it reported 406 warnings. However, since Phantm does not provide call context of reported problems, inspecting these warnings in application of NOCC size is laborious if not impossible. Moreover, high false-positive rate of 93% even in the benchmark application makes the output of Phantm in larger applications almost useless[3].

Warnings (#)	54
False positive rate (%)	76
Time (s)	238

Table 7.4: Analysis results for WeVerca for NOCC email client comprising of 15605 lines of code.

Identified Problems

Out of 54 alarms, 13 alarms correspond to real problems. We now describe a selection of them:

1) Sending tainted value to the browser. WeVerca reported one case where a value that can be influenced by the input is sent to the browser without being sanitized. The simplified version of the corresponding code is:

```
$domainnum = $_REQUEST['domainnum'];
$_SESSION['nocc_domainnum'] = $domainnum;
print('<div><input type="hidden" name="
    saved_domainnum" value="'.$_SESSION['
    nocc_domainnum'].'" /></div>');
```

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

[3] Phantm allows invoking static analysis from a particular runtime state [34]. This greatly reduces false positive rate. However, the application is analyzed only partially.

The value from the request (`$_REQUEST['domainnum']`) is propagated to the index `$_SESSION['nocc_domainnum']` and then sent to the browser. Note that as this flaw allows performing cross site scripting attacks, it is very dangerous. In this case, the attacker can manipulate the request (e.g., by sending a victim a link that represents manipulated request) to `$_REQUEST['domainnum']` contain a malicious JavaScript. If the victim clicks on the link, the JavaScript in `$_REQUEST['domainnum']` is sent to the victim's browser and is executed there.

2) Opening file specified by tainted value. WeVerca reported two cases where values that can be influenced by the input are used to specify the file to be opened. The simplified version of the corresponding code is:

```
$_SESSION['nocc_user'] = $_REQUEST['user'];
$server = $_REQUEST['server'];
$user_key = $_SESSION['nocc_user'] . '@' .
  $_SESSION['nocc_domain'];
$filename = $conf->prefs_dir . '/' . $user_key .
  '.filter';
$file = fopen($filename, 'r');
```

Values from the user input (`$_REQUEST['user']` and `$_REQUEST['server']`) are propagated to indices `$_SESSION['nocc_user']` and `$_SESSION['nocc_domain']`. Finally, `$user_key` is used to create `$filename`, which is used to specify the file to be opened. Note that by manipulating `$_REQUEST['user']` and `$_REQUEST['server']` any file with character `'@'` in its name of or its path and extension `filter` located in arbitrary directory in the server can be opened.

3) Calling a function with undeclared argument. WeVerca detected one case where a function was called with an argument that is not declared. The simplified version of the corresponding code is:

```
function save_session() {
  ...
}
save_session($argument);
```

The function `save_session` has no argument declared. Since in PHP, the function to be called is given only by its name (PHP does not support function overloading), PHP calls the correct function and ignores the argument. However, we believe that such calling a function with superfluous argument is a bad practice.

4) Superfluous implicit conversions. WeVerca detected several cases of superfluous implicit conversions. For example, it detected that built-in function `set_magic_quotes_runtime` is used with integer argument instead of boolean. The triggered implicit conversion of integer to boolean results in correct result; however, we believe that implicit conversions should be avoided when possible.

False Alarms

WeVerca reported 41 false alarms. While the number of false alarms is high, they have relatively small number of causes. We now describe all these causes of reported false alarms.

1) Imprecisely modeled built-in functions. WeVerca reported 21 false alarms due to imprecise modeling of return value of built-in functions. As an example, here is the simplified version of the code corresponding to false positive due to imprecise modeling of the function `imap_mime_header_decode`:

```
$source = imap_mime_header_decode($header);
for ($j = 0; $j < count($source); $j++ ) {
    $element_charset = ($source[$j]->charset == '
        default') ? detect_charset($source[$j]->text)
        : $source[$j]->charset;
}
```

While the function `imap_mime_header_decode` returns array of objects with fields `text` and `charset`, WeVerca models the return value just by `AnyArray`. Accessing `AnyArray` yields `AnyValue` and finally, accessing fields `text` and `charset` on `AnyValue` causes reporting a false alarm.

2) Widening of arrays. WeVerca reported 10 false alarms due to widening of arrays. To guarantee the termination of the analysis, the current implementation of heap analysis use widening of arrays and objects. If the the number of indices of some array infinitely grows, the array is replaced with `AnyArray`. Since adding or removing indices to `AnyArray` has no effect—it results again in `AnyArray`, the computation eventually converges. However, this widening is neither precise nor sound—as the content of the array is lost, it can lead both to false alarms and also to missed errors. The simplified version of the corresponding code is:

```
$structure = imap_fetchstructure($conn, $msgnum);
$num_parts = count($structure->parts);
```

```
$parts = array();
for ($i = 0; $i < $num_parts; $i++) {
    $part[$i] = new NOCC_MailPart($structure, $i);
}
...
foreach ($parts as $mailPart) {
    $mailPart->getPartStructure();
}
```

Since the built-in function `imap_fetchstructure` is modeled just by types, it returns `AnyObject`. Accessing a field of `AnyObject` returns `undefined` and `AnyValue`. In the former case, WeVerca reports a false alarm that possibly undefined object property is accessed. Next, the variable `$parts` is initialized with an empty array. In the `for` cycle, the array is filled with instances of `NOCC_MailPart` until the array is widened to `AnyArray`. Next, accessing the indices of the array in the `foreach` cycle yields `AnyValue` and finally a false alarm reporting that the method `getPartStructure` is not defined is reported. Note that in this example, due to widening of the array, the method `getPartStructure` is not analysed and the analysis can also miss errors.

To solve this problem, we want to employ widening that soundly overapproximates values stored in the array to be widened.

3) Unmodeled assumption of built-in function. WeVerca reported 6 false alarms that unsanitized value from user input can be sent to the browser. In all these cases, the input was correctly sanitized by testing whether the input value is equal to the name of some key in given array. The simplified version of the corresponding code is:

```
$themeName = $_REQUEST['theme'];
if (array_key_exists($themeName, $this->_themes))
    $_SESSION['nocc_theme'] = $theme_name;
else
    $_SESSION['nocc_theme'] = 'standard';
echo '<img src="themes/' . $_SESSION['nocc_theme'] .
   '/img/new.png" alt="" />';
```

In the `if` branch a value from the input is assigned to the variable `$_SESSION['nocc_theme']`. However, the condition of the branch implies that the input value is equal to the name of some key in the array `$this->_themes` containing all valid themes. In the `else` branch, the variable `$_SESSION['nocc_theme']` is assigned a constant value. That is, the value in the variable `$_SESSION['nocc_theme']` is valid in all cases and it can be safely

sent to the browser.

WeVerca does not model assuming the effects of the function `array_key_exists` and it is thus not able to deduce that the input is sanitized in the `if` branch.

4) Path-insensitivity. WeVerca reported two false alarms that a single file can be incorrectly included more than once due to path-insensitivity when modeling the function `require_once`. In PHP, this function checks whether a specified script has been already included and if not, it includes (interprets) the script. It can be thus modeled as:

```
if (!included($file)) require($file);
```

The included script may contain arbitrary code, but in most cases, it contains just declarations of functions, classes, and constants. In PHP, functions, classes, and constants are declared at runtime, but attempts to redeclare these result in runtime error. When trying to find these errors, the analyzer reported many false alarms[4] due to path-insensitivity in the following cases:

```
$input = $_GET['input'];
if ($input)
  require_once('nocc_mailaddress.php');
require_once('nocc_mailaddress.php');
```

Joint point of the `if` statement discards the information that the script `'nocc_mailaddress.php'` was certainly included and WeVerca includes the script twice.

The same way as in the case of false alarms due to path-insensitivity in the benchmark application, these false alarms can be detected by the technique of path-sensitive validation of alarms proposed in [24].

5) Imprecise modeling of read accesses to abstract strings. WeVerca reported an access to possibly uninitialized string offset. The simplified version of the corresponding code is:

```
for ($i=0; $i<strlen($string); $i++) {
  $string[$i] = $string[$i]^$key;
}
```

The variable `$i` ranges over initialized offsets of the variable `$string`. However, the content of the variable `$string` is not statically known and

[4] In order to not overload user with too many false alarms, we switched off individual alarms reporting attempts to redeclare constants, functions, and classes. Instead, we just report that a script can be included more than once using `require_once`.

WeVerca abstracts it using AnyString abstract value. Since WeVerca models the result of call to function strlen on AnyString using AnyInteger abstract value, it reports an access to possibly uninitialized string offset. That is, the false alarm is caused by imprecise modeling the correlation between the length of the string being argument of the function strlen and the result of the function.

6) Imprecise modeling of assumptions. In one case, WeVerca reported sending possibly uninitialized data to the browser due to imprecise modeling of assumptions. When processing assumptions, WeVerca refines only data stored in variables. That is, data stored in array indices and object properties are not refined. The simplified version of the corresponding code is:

```
if (!isset($_SESSION['nocc_theme'])) {
    $_SESSION['nocc_theme'] = 'standard';
}
echo $_SESSION['nocc_theme'];
```

While the assumption of the if branch implies that the index $_SESSION['nocc_theme'] is undefined, the assumption of the else branch implies that the index is defined. However, in former case, the index is defined by the assignment. That is, the index must be defined after the join of the if statement. However, since the data are stored in array index, WeVerca is not able to infer that the index cannot be undefined in the else branch and reports the false alarm.

7.3 Summary of Chapter 7

To summarize the results, the evaluation performed on synthetic PHP code shows that the heap analysis implemented as a part of our framework scales well even when complex data accesses are involved. Merging analysis states is fast even for analysis states with complex structures. Moreover, merging analysis states scales almost linearly with number of variables modified in merged branches. The scalability of the current implementation is limited by the fact that it does not share any data between the states. For future work, we want to eliminate this limitation by employing data sharing.

We showed that WeVerca can scale even for real applications. We used WeVerca to conduct two case studies analyzing over 16,000 lines of PHP code.

The results for the first case study (benchmark application) are impressive. WeVerca outperforms state-of-the-art tools in both error coverage and false positive rate while achieving good analysis time.

For the second case study (NOCC email client), WeVerca found one previously unknown security flaw, which allows the attacker to perform cross site scripting attacks and two less serious security flaws. Moreover, WeVerca found 10 problematic code fragments that we consider a bad practice. While WeVerca reported 41 false alarms, these false alarms were caused by relatively few limitations of the tool. These are imprecise modeling of built-in functions, imprecise (and unsound) widening of arrays, path-insensitivity, imprecise modeling of read accesses to abstract strings, and imprecise modeling of assumptions. In our future work, we want to eliminate these limitations.

8

Conclusion and Future Work

Looking back at the goals from Section 3, i.e.:

- **G1** *Design heap analysis for dynamic languages.*
- **G2** *Generically define interplay of heap and value analyses for dynamic languages.*
- **G3** *Design framework for static analysis of dynamic languages.*

the contribution of this thesis can be summarized as follows:

Fulfilling the goal G1 We created heap analysis modeling associative arrays and prototype objects and backed it with full formalization. Since prototype objects can be modeled using associative arrays, we described the analysis in terms of associative arrays.

The analysis tackles the following challenges in static analysis of associative arrays: (1) Indices are not declared—if an updated index exists, it is overwritten, otherwise it is created. (2) Indices can be accessed using arbitrary expressions, which can yield even statically unknown values. Consequently, the set of indices employed for an array is not evident from the code. (3) Specifically for multidimensional arrays, updates cannot be decomposed. The reason is that updates create indices if they do not exist and initialize them with empty arrays if also further dimensions are updated; on contrary, read accesses do not.

Since existing heap analyses for dynamic languages address just challenges (1) and (2), the main contribution of our heap analysis is that we address all of them.

We solved the problem of updates to statically unknown array indices by employing special index called *unknown field*, which is contained in each array and stores information from such updates. To represent multidimensional arrays, all indices (including unknown fields) can contain a set of values and also can point to another associative array—the next dimension.

To take into account the semantics of non-decomposable multidimensional updates, an update is represented using a list of expressions, where each expression specifies an access to one dimension of the array. The update starts in the first level. Then, at each level indices corresponding to values of an expression that belongs to the current level, are followed. If the expression yields a statically unknown value, all indices (including the unknown field) are followed. If the expression yields values for which there do not exist corresponding indices in the array, these indices are created and all data that could be assigned to new indices using previous statically unknown updates are copied to these indices. Then, the traversal continues using also new indices.

Finally, we defined how multidimensional associative arrays with unknown fields are joined. Here, the challenge was to determine the set of indices of the resulting array—due to the unknown fields and semantics of accesses to associative arrays, the resulting array may contain indices that are not present in any array being merged. After the set of indices of the resulting array is determined, indices in the input arrays can be merged pairwise.

While we focused on soundness and precision of the analysis, we showed that prototype implementation of the analysis scales to real-world programs.

Fulfilling the goals G2 and G3 We introduced the framework for static analysis of dynamic languages that allows defining end-user static analyses independently of dynamic features. This is possible because: (1) The framework defines how heap and value analyses interplay. This allows value analysis to automatically access variables, array indices, and object properties. (2) The framework coordinates resolving dynamic features. It defines which information must value analysis provide for resolving dynamic features (i.e., to compute control flow and resolve dynamic data accesses), and it allows user to define how these features are resolved. Since our implementation of the framework provides default implementation of the value analysis and provides default definitions of resolving dynamic features, these features can be resolved fully automatically. (3) Finally, the framework allows defining additional value analyses tracking arbitrary values.

The interplay of heap and value analyses is possible because the heap analysis tracks the shape of the heap and approximates concrete locations with heap identifiers while the value analysis tracks values on heap identifiers. The fundamental challenge when defining the interplay was to take into account that array indices and object properties can be added at runtime by dynamic updates—targets of these updates can be even statically unknown.

To be able to capture this semantics, heap analysis can materialize heap identifiers both during the assignment and join operation. These changes are propagated to value analysis. The propagation is done automatically by the framework using assignment transfer function for the value analysis. As a result, various heap analyses can be combined with arbitrary (non-relational) value analyses and these analyses can be defined independently. Finally, we also showed how the interplay can be extended to also support relational value analyses.

The control flow is captured in the intermediate representation and since it is not known before the analysis, the framework builds intermediate representation during the analysis. To provide the framework values necessary for computing control flow and values for resolving dynamic index and property accesses, we defined a lattice structure for the first-phase value analysis. The lattice structure provides information about values of PHP primitive types that a heap identifier (i.e., variable, array index, and object property) can store at a program point. Because of dynamic type system, a heap identifier can store values of more types at a single program point.

As a proof-of-the-concept, we designed WeVerca, a static analysis framework for PHP applications. To enable sound and precise computing of control-flow graph and resolving accesses to associative arrays and objects, we implemented heap analysis described in Chapter 4 and also value analysis modeling PHP primitive types, type conversions, native operators, and library functions. Moreover, we used the framework to implement additional value analysis—taint analysis for detection of security problems. We also used the framework to create a tool for PHP code analysis integrated to Eclipse IDE.

8.1 Open Issues and Future Work

As a future work, we plan to improve the scalability and precision of the analysis framework, and we plan to use the framework to define new analyses. In particular, we want to:

Enhance scalability. We carried out case studies on real, but rather small programs. We were not able to analyze large programs because of limited scalability of the implementation of our static analysis framework. However, there are several opportunities for improving the scalability.

First, the implementation still stays on a prototype level and can be optimized. For example, the representation of abstract states employs

almost no sharing, we implemented only full context-sensitivity, and we model most library functions just using type information.

Second, our analysis is global—it analyses the whole program. This limits the scalability especially when libraries and frameworks are used. In some cases, the application contains parts that are unsuitable to analyze with the same abstractions as the rest of the application, e.g., parts of a content management system that perform numerical calculations. In these cases, it would be useful to analyze these parts separately using different abstractions and then compose the analyses of these parts to get the information on the whole program [10]. In some cases, it could be useful to analyze shared functions once, create summaries of these functions, and then reuse these summaries [57, 14]. Other possibility is to explicitly model individual functions (e.g., using pre- and post- conditions) or whole parts of the applications (e.g., frameworks) [16, 47]. Next, combining static analysis with dynamic analysis would allow analyzing programs for specific inputs thus enabling to analyze even larger programs, but sacrificing soundness.

Eliminate known causes of false alarms. When conducting the case studies, we discovered several limitations of the current implementation of our static analysis framework causing false alarms to be reported. The most important limitations are imprecise modeling of some built-in functions, imprecise modeling of assumptions, and imprecise widening of arrays and objects. In our future work, we want to eliminate these limitations.

Enhance precision. In further case studies, we want to investigate if there is a need for even higher precision. For example, to prove more program properties, we could extend our framework to support relational abstract domains. While our framework now supports only non-relational domains, as discussed in Section 5.9, extending it to relational domains is straightforward. Next, string component of our value analysis abstract state could be replaced with more precise representation, e.g., string automaton [59]. Another possibility to gain more precise results is path-sensitive validation of analysis warnings [24].

Create more lightweight heap analysis. When designing our heap analysis, we focused on soundness and precision. However, for some applications such as code navigation and bug finding, unsound but fast heap analysis would be more usable. For these applications, more lightweight and scalable heap analysis should be created.

Make the analysis incremental. When using static analysis during the development, the analysis results should be updated frequently to reveal errors as soon as possible. Since in this case, the portion of application that was modified since the last time the program was analyzed is usually be small, making the analysis incremental can significantly shorten the analysis time [51, 22, 37].

Use the framework to build programming tools. Finally, we want to use our framework to define more value analyses and use these analyses to build tools, e.g., for error detection, code refactoring, and code optimization.

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *SAS'06: Proceedings of the 13th International Conference on Static Analysis*, Lecture Notes in Computer Science, pages 221–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivancic, Ou Wei, and Aarti Gupta. Slr: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS'08: Proceedings of the 15th International Conference on Static Analysis*, Lecture Notes in Computer Science, pages 238–254. Springer-Verlag, Berlin, Heidelberg, 2008.

[4] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[5] Constantinos Bartzis and Tevfik Bultan. Widening arithmetic automata. In *CAV '04: 16th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 321–333, Berlin, Heidelberg, 2004. Springer-Verlag.

[6] Jan Benda, Tomas Matousek, and Ladislav Prosek. Phalanger: Compiling and running php applications on the microsoft.net platform, 2006.

[7] Paul Biggar and David Gregg. Static analysis of dynamic scripting languages. `http://paulbiggar.com/research/wip-optimizer.pdf`, 2009.

[8] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05: Proceedings*

of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, pages 147–163, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 296–310, New York, NY, USA, 1990. ACM.

[10] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 159–178, London, UK, UK, 2002. Springer-Verlag.

[11] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, New York, NY, USA, 2002. ACM.

[12] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-sensitive dataflow analysis with iterative refinement. In *SAS'06: Proceedings of the 13th International Conference on Static Analysis*, Lecture Notes in Computer Science, pages 425–442. Springer-Verlag, Berlin, Heidelberg, 2006.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, New York, NY, USA, 2008. ACM.

[14] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI '11: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 567–577, New York, NY, USA, 2011. ACM.

[15] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 118–128, New York, NY, USA, 2007. ACM.

[16] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS '10: Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*,

Lecture Notes in Computer Science, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. Tool-supported refactoring for javascript. In *OOPSLA '11: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 119–138, New York, NY, USA, 2011. ACM.

[18] Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In *OOPSLA '13: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 323–338, New York, NY, USA, 2013. ACM.

[19] Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI'05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 302–321, Berlin, Heidelberg, 2014. Springer-Verlag.

[20] Zhoulai Fu. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for java. In *VMCAI '05: Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 282–301, Berlin, Heidelberg, 2014. Springer-Verlag.

[21] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In *TACAS '04: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 512–529, Berlin, Heidelberg, 2004. Springer-Verlag.

[22] Salvatore Guarnieri and Benjamin Livshits. Gulfstream: Staged static analysis for streaming javascript applications. In *WebApps'10: Proceedings of the 2010 USENIX Conference on Web Application Development*, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[23] David Hauzar and Jan Kofron. Hunting bugs inside web applications. Technical report, Department of Informatics, Karlsruhe Institute of Technology (presented in FoVeOOS '11), 2011.

[24] David Hauzar and Jan Kofron. On security analysis of php web applications. In *COMPSACW '12: Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 577–582, Washington, DC, USA, 2012. IEEE Computer Society.

[25] David Hauzar and Jan Kofron. WeVerca: Web verification for php. In *SEFM '14: Proceedings of the 12th International Conference on Software Engineering and Formal Methods*, Lecture Notes in Computer Science, Berlin, Heidelberg, 2014. Springer-Verlag.

[26] David Hauzar, Jan Kofroň, and Pavel Baštecký. Data-flow analysis of programs with associative arrays. In *ESSS '14: Proceedings of the 3rd International Workshop on Engineering Safety and Security Systems*, Electronic Proceedings in Theoretical Computer Science, pages 56–70. Open Publishing Association, 2014.

[27] 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: Proceedings of the 13th International Conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[28] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1930–1937, New York, NY, USA, 2009. ACM.

[29] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA 2012: Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 34–44, New York, NY, USA, 2012. ACM.

[30] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS'09: Proceedings of the 16th International Static Analysis Symposium*, volume 5673 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, August 2009. Springer-Verlag.

[31] 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.

[32] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *J. Comput. Secur.*, 18(5):861–907, September 2010.

[33] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Fse '10: Phantm: Php analyzer for type mismatch. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 373–374, New York, NY, USA, 2010. ACM.

[34] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *RV'10: Proceedings of the First International Conference on Runtime Verification*, Lecture Notes in Computer Science, pages 300–314, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL '11: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, New York, NY, USA, 2011. ACM.

[36] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL '13: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 385–398, New York, NY, USA, 2013. ACM.

[37] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *CC'13: Proceedings of the 22nd International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[38] Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS'10: Proceedings of the 17th International Conference on Static Analysis*, Lecture Notes in Computer Science, pages 71–99, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM.

[40] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES '06: Proceedings*

of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems, pages 54–63, New York, NY, USA, 2006. ACM.

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

[42] Phalanger. `http://phalanger.codeplex.com/`, 2014. [Online; accessed 23-June-2014].

[43] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *PLDI '13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–174, New York, NY, USA, 2013. ACM.

[44] Viliam Simko, David Hauzar, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Verifying temporal properties of use-cases in natural language. In *FACS '11: Proceedings of the 8th International Symposium on Formal Aspects of Component Software*, Lecture Notes in Computer Science, pages 350–367, Berlin, Heidelberg, 2011. Springer-Verlag.

[45] Viliam Simko, David Hauzar, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Formal verification of annotated textual use-cases. In *Computer Journal (submitted)*, 2014 (current status: minor revision).

[46] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, October 2006.

[47] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA '11: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 1053–1068, New York, NY, USA, 2011. ACM.

[48] 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.

[49] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. Information flow analysis via path condition refinement. In *FAST'10: Proceedings of the*

7th International Conference on Formal Aspects of Security and Trust, Lecture Notes in Computer Science, pages 65–79, Berlin, Heidelberg, 2011. Springer-Verlag.

[50] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[51] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Fase'13: Andromeda: Accurate and scalable security analysis of web applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 210–225, Berlin, Heidelberg, 2013. Springer-Verlag.

[52] Arnaud Venet. Towards the integration of symbolic and numerical static analysis. In *VSTTE: Proceedings of Verified Software: Theories, Tools, Experiments*, Lecture Notes in Computer Science, pages 227–236, Berlin, Heidelberg, 2005. Springer-Verlag.

[53] W3Techs. `http://w3techs.com/`, 2009. [Onlne; accessed 1-December-2009.

[54] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, New York, NY, USA, 2007. ACM.

[55] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[56] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013: Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346, New York, NY, USA, 2013. ACM.

[57] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[58] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS'10: Proceedings of the 16th*

International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, pages 154–157, Berlin, Heidelberg, 2010. Springer-Verlag.

[59] Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN '08: Proceedings of the 15th International Workshop on Model Checking Software*, Lecture Notes in Computer Science, pages 306–324, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The hiphop compiler for php. In *OOPSLA '12: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 575–586, New York, NY, USA, 2012. ACM.

[61] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *ICSE '13: Proceedings of the 2013 International Conference on Software Engineering*, Lecture Notes in Computer Science, pages 652–661, Piscataway, NJ, USA, 2013. IEEE Press.