Weverca

Web verification tool for PHP

Programmer documentation

Contents

1 Introduction 5
1.1 Analysis of dynamic languages 5
1.2 PHP 5
1.3 Challenges of analysing PHP 5
1.3.1 Variables, arrays, and objects 6
1.3.2 Dynamic accesses 6
1.3.3 Explicit Aliasing 7
1.3.4 Dynamic Class Definitions 7
1.3.5 Dynamic Calls and Dynamic Accesses 8
1.3.6 Constants 8
1.3.7 Comparison to other languages 8
1.4 Related tools 9
1.5 Goals of the project 9
2 Development 11
2.1 Time table 11
2.2 Team members 11
2.3 Firing Michal Staša 11
3 Developer documentation 12
3.1 Static analysis of PHP 12
3.1.1 Analysis workflow 12
3.1.2 Abstract syntax tree 12
3.1.3 Intra-procedural Control-flow graph 12
3.1.4 Inter-procedural Control-flow graph 13
3.1.5 Principles of static analysis 13
3.1.6 Static analysis of PHP 15
3.2 Requirements 18
3.3 Architecture 18
3.4 Connection to Phalanger 20
3.5 Control-flow graph 21
3.5.1 Examples 23
3.6 Metrics 28
3.6.1 User-implemented metrics 29

3.6.2 Class presence ... 29
3.6.3 MySQL or other SQL functions usage ... 29
3.6.4 Dereference with double $ ... 29
3.6.5 Maximal depth of method overriding ... 29
3.6.6 Maximum inheritance depth ... 30
3.6.7 Dynamic calls and object creations ... 30
3.6.8 Class alias ... 30
3.6.9 Magic methods ... 30
3.6.10 Dynamic inclusion ... 30
3.6.11 Number of lines and source files ... 30
3.6.12 Superglobals ... 30
3.6.13 Function/type declaration inside function body ... 31
3.6.14 Duck typing ... 31
3.6.15 Passing variable by reference at call side ... 31
3.6.16 Autoload ... 31
3.6.17 Class coupling ... 31
3.6.18 Functions coupling ... 31
3.6.19 Eval ... 31
3.6.20 Session functions ... 31
3.6.21 References ... 32
3.7 Framework for analysis ... 32
3.7.1 Forward analysis ... 32
3.7.2 Program point ... 32
3.7.3 Program point graph ... 33
3.7.4 Assumptions ... 35
3.7.5 Try blocks ... 36
3.7.6 Extension branches ... 36
3.7.7 Sharing program point graphs ... 37
3.7.8 Fixpoint computation ... 38
3.7.9 Second phase analysis ... 39
3.8 Memory models ... 40
3.8.1 Virtual Reference memory model ... 40
3.8.2 Copy memory model ... 45

3.9 Function resolver 56
3.9.1 Native analyzers 57
3.9.2 Function hints 58
3.10 Expression resolver 58
3.10.1 Overview 58
3.10.2 Conversions 58
3.10.3 Unary and n-ary operations 59
3.10.4 Binary operations 59
3.10.5 Variable resolving 60
3.10.6 Creating new values 60
3.10.7 Type declarations 60
3.10.8 Static variable storage 61
3.10.9 Global constant storage 61
3.10.10 Class constant storage 61
3.10.11 Foreach 61
3.10.12 Special constructs and build-in functions 61
3.11 Flow resolver 62
3.11.1 Overview 62
3.11.2 How does it work 62
3.11.3 Exceptions 63
3.11.4 Includes 63
3.11.5 Eval 63
3.11.6 Future works 64
3.12 Adding support for new PHP features 64
3.13 Web 64
3.13.1 Project settings 64
3.13.2 Debugging of the project 65
3.13.3 Deployment 65
3.13.4 Future works 65
4 Conclusion 66
5 References 67

1 Introduction

1.1 Analysis of dynamic languages

Languages with dynamic constructs such as dynamic type system, virtual methods, reflection, dynamic data structures, provides flexibility and accelerates the development, in particular, the development of web applications. However, these languages shift more work to development tools such as tools for code analysis, error discovery, code refactoring, code optimization, code navigation, and code autocompletion.

For most of these tools, static program analysis is a necessary prerequisite. Static program analysis computes information about a program valid for all its possible executions. This includes, e.g., information about control-flow of the program, information about values and types of variables in given program point.

Unfortunately, dynamic features pose major challenges here. For instance, any interprocedural data-flow analysis needs to track types of variables to determine targets of virtual method calls. This becomes even more important in the case of languages with dynamic type system, where types of variables can be completely unspecified. 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 computed at run-time. 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 first class names, e.g., variables. Interprocedural data-flow analysis thus furthermore needs to track values of variables. This happens relatively often, e.g., in web applications, which manipulate a lot of input.

1.2 PHP

PHP is the most common programming language used at the server side of web applications. It features many dynamic constructs such as:

- Dynamic includes
- Indirect method or function calls
- Indirect variable use
- Variable aliasing
- Dynamic object and function declaration
- Conditionally defined global constants, constants are not really constant

1.3 Challenges of analyzing PHP

We aimed to create a tool that can process and analyze all possible dynamic PHP constructs. In following example, we show how some of dynamic features impacts the data-flow analysis and influence the requirements on memory models.

```
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;
```

1.3.1 Variables, arrays, and objects

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

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

1.3.2 Dynamic accesses

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

An update can involve more than one element and can be statically unknown. The update at line 4 is statically unknown and thus may or may not influence accesses at lines 5, 7, 8, 9, and 15. 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.

The semantics of the PHP object model is similar to the semantics of associative arrays. Object's 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 first-class names. Objects can 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.

1.3.3 Explicit Aliasing

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

Unlike C++, in PHP each variable, index, and property can be aliased and later un-aliased from its previous aliases and become an alias of a new element. As an example, the statement at line 16 un-aliases `$arr[2]` from its previous aliases. Moreover, a variable can be an alias of another variable only at some paths to a given program point, e.g., if it is made an alias in a single branch of an if statement.

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

1.3.4 Dynamic Class Definitions

In PHP, a definition of the class can be put almost everywhere in the code, e.g., inside a function and in a conditional branch. The definition is then resolved at runtime. If the definition of a class is reached when interpreting the code, the class is defined. If the class is already defined, it cannot be redefined. That is, there must be at most one definition of a single class in a single program path. However, there can be program paths with different definitions of a single class. Example:

```
if($_POST["a"]=="something")
{
	class x
	{
		public $a=4;
		function a(){}
		function b(){}
	}
	$p="a";
}
else
{
```

```
	class x
	{
		public $v=4;
		function a(){}
		function b(){}
	}
	$p="b";
}
$x=new x();
$x::$p();
```

The last line contains static method call on a class corresponding to object stored in the variable `$x`. The variable $x contains object of class x, but type x has multiple meanings.

1.3.5 Dynamic Calls and Dynamic Accesses

In PHP, the name of the function or method to be called can be specified using an arbitrary expression. An example of dynamic call is in the last line of the previous example. Variable `$p` has possible values string “a” or “b” and the name of the method to be called is determined by variable.

Note that not only method name can be determined by a variable value, but also variable name can be determined by other variable value. The name of the class can be also determined dynamically, e.g., if the variable `$x` in the previous example contained a string, e.g., “o”, this call would be treated as static call on class o.

1.3.6 Constants

Another example:

```
if($_POST["a"]=="something")
{
	define("a",0);
else
{
	define("a",1);
}
echo a;
```

In this example constant a has possible value 0 or 1. That means that constants are not really constant and they cannot be stored in some global table, but in memory model as special variables.

1.3.7 Comparison to other languages

It is worth mentioning that a plenty of other languages, especially those connected with the development of web applications, have the support for associative arrays-like data structures. These languages includes Javascript, Python, Ruby, etc. Moreover, to ease the

development, some “ordinary” programming languages emulate some of these features and offer the developer API behaving in a similar way.

1.4 Related tools

Pixy [1] is an open-source tool for detection of taint-style vulnerabilities in PHP 4. It involves a flow-sensitive, interprocedural, and context-sensitive data flow analysis along with literal and alias analysis to achieve precise results. The main limitations of Pixy include a limited support for statically-unknown updates to associative arrays, ignoring classes and the eval command, and limited support for aliasing and handling file inclusion, which all represent principal differences from programming languages such as Java and C. Alias analysis introduced in Pixy incorrectly models aliasing between arrays and array indices. Web applications use associative arrays and objects extensively, thus we believe that this is an essential limitation. Importantly, Pixy does not perform type inference, which also limits its precision and soundness.

Stranger [2] is an automata-based string analysis tool for PHP, which is built upon Pixy. It adds a more precise string manipulation techniques that enable the tool to prove that an application is free from attack patterns specified as regular expressions.

Phantm [3] is a PHP 5 static analyzer for type mismatch based on data-flow analysis; it aims at detection of type errors. It combines run-time information from the bootstrapping phase of an application and static analysis when instrumentation using this information is used. To obtain precise results, Phantm is flow-sensitive, i.e., it is able to handle situations when a single variable can be of different types. However, they omit updates of associative arrays and objects with statically-unknown values and aliasing, which can lead to both missing errors and reporting false positives.

1.5 Goals of the project

To implement any interprocedural static analysis for PHP (as well as for any other dynamic language), one needs to combine the end-user analysis with other analyses necessary just to allow the static analysis, e.g., type analysis and literal analysis and needs to correctly read data from and write data to built-in data structures, such as multi-dimensional associative arrays and objects. As there are many choices of implementing these necessary aspects that affect the scalability and precision of the resulting tool, e.g., the choice of context sensitivity, the choice of abstract domains, and the way in which library functions are modeled, and there are no means for explicitly separating these from the end-user analyses, tools implement all from scratch and combine these necessary aspect with end-user analyses in its own way. Consequently, implementations of static analyses become either complex or imprecise. In order to tackle these problems, the project aims at the following goals:

- The first goal is to design a framework that makes it possible to implement type analysis, literal analysis and the modeling of built-in data structures independently of the end-user static analysis.
- The second goal is to provide default implementations of all analyses that are necessary to automatically resolve dynamic features and allow to run end-user analyses. The analyses should be precise yet scalable.

- The third goal is to implement static taint analysis as end-user analysis as the proof-of-the-concept.

2 Development

2.1 Time table

2013

January	analyzing possible solutions
March - April	adjusting Phalanger parser and building control flow graph creating framework for computing metrics
April	started implementation of analysis framework
April to July	metric implementation
05.04. 2013	project was officially started
May	copy memory model implementation started virtual reference model implementation started
Jun	started implementation of native analyzers
July	started implementation of expression resolver and flow resolver
September	started implementation of object model and function resolver finished implementation of native analyzers
December	started implementation of web implementing warning outputs second phase analysis implementation

2014

Middle of February	finished implementing flow resolver, expression resolver function resolver, memory models, object model, analysis framework, web
End of February	testing, bug fixing and creating documentation

2.2 Team members

David Škorvaga
Marcel Kikta
Matyáš Brenner
Michal Staša
Miroslav Vodolán
Pavel Baštecký

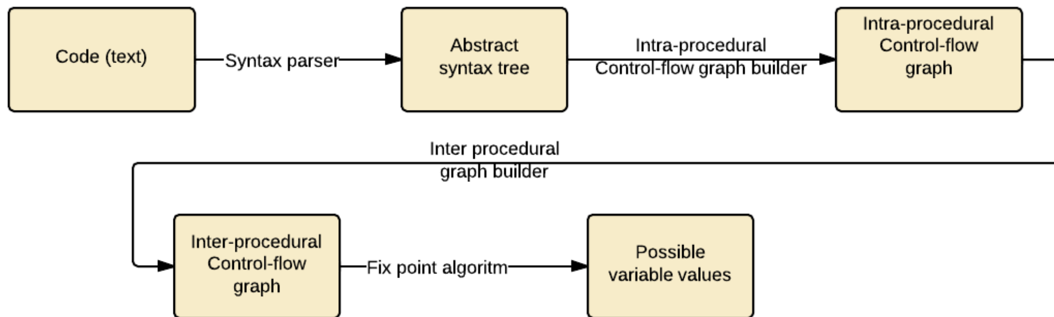
2.3 Firing Michal Staša

In August 2013 we decided to fire Michal from the team, because he didn't engage in work, and didn't start with any implementation. His actions had a bad influence on motivation of the other team members. The decision to fire him was unanimous.

3 Developer documentation

3.1 Static analysis of PHP

3.1.1 Analysis workflow



On the image above the workflow of static analysis is shown. Source code as text is parsed by syntax parser and it outputs an abstract syntax tree. Intra-procedural control-flow graph is created from the abstract syntax tree. This control flow graph is transformed into inter-procedural control-flow graph, which is used as an input of fixpoint algorithm. This algorithm outputs possible variable values for every place of the input source code.

In following chapters we explain data structures and algorithms used in static analysis.

3.1.2 Abstract syntax tree

Abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of a source code written in a programming language. Each node of the tree represents a language element from the source code. The syntax tree is abstract and it is not representing all syntax elements e.g. parentheses.

3.1.3 Intra-procedural Control-flow graph

Basic block is a list of statements or instructions, which doesn't contain any jump statements or jump instructions.

Control-flow graph (CFG) is a graph representation of computation and control flow in the program. Nodes in this graphs are basic blocks. Edges represents possible flow from the end of one block to the beginning of the other.

Functions and class definitions are inserted into the graph without any processing. They are processed in the following phases.

The main difference between intra-procedural CFG and AST is that CFG represents flow in the program and AST represents program in the unchanged order.

3.1.4 Inter-procedural Control-flow graph

Intra-procedural CFG described above is limited to scope of a single function. However for describing flow of the whole program it is needed to track flow across function and method calls. Therefore edges from call statements into corresponding declarations are needed.

Unfortunately it is not always possible to determine which declaration belongs to the call. It is caused by language constructs that allow choosing function for the call according to runtime information. For strongly typed languages the construct can be a virtual call. For dynamic languages there are even more options like conditional function declarations etc.

Some workarounds that choose candidate functions for the call according to type inheritance can be used in strongly typed languages. However for purposes of this work the workaround is not usable because of the lack of the strong typing of PHP. These facts prevents the analyzer to build complete inter-procedural CFG before additional information from analysis are available. Therefore our implementation of inter-procedural CFG has to be able to dynamically change its structure. The implementation of the graph is called Program Point Graph (PPG) for purposes of this work and it will be described in detail in implementation chapters.

3.1.5 Principles of static analysis

Static analysis provides ability to gather information about programs without the need to execute them. That is independently of their inputs. Usual answers which static analysis can provide are related to possible values of variables or determining superfluous expressions in the code. This kind of information can be used for compiler optimizations and providing hints by development tools.

The way static analysis works is based on computing information about program environment for each node of a control flow graph. In each program point there are two sets of facts known about the environment. The first set that is called input set is used for facts that are known before execution of statement in the program point. The output set contains the facts known after the execution. The transition between the sets is defined by transfer function reflecting the semantics of the represented statement.

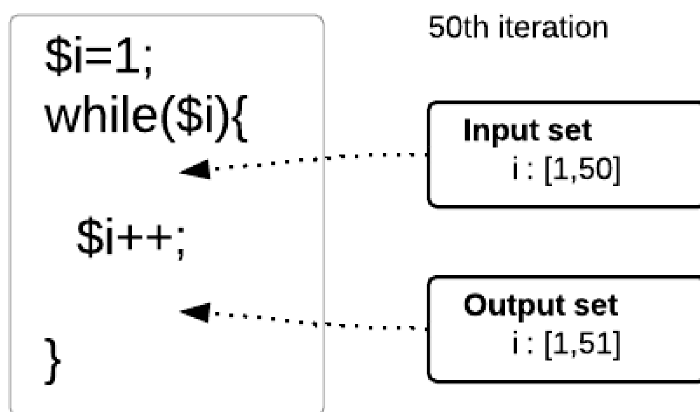
Two directions of analysis can be distinguished. Forward analysis, which is usually used for simulation of runtime behavior of the program, is computed in the same direction as the program flow. The other analysis is backward and is handful, e.g., for situations where unused program variables are searched. This analysis is computed in the opposite direction of the program flow.

The goal of the analysis is to compute a state called fixpoint. In this state it is required that input and output sets contains facts such that using transfer function on any of program points will not add any new facts about program environment.

The algorithm used for finding fixpoint is called worklist algorithm. It starts with the list containing entry program point. Then it takes a program point from the list and update its output by using the transfer function. If the output set of the program point is changed in comparison to previous state, the children of the program point in control flow graph are added to the list. These steps are repeated until there is no available program point in the list. In this case fixpoint is successfully found.

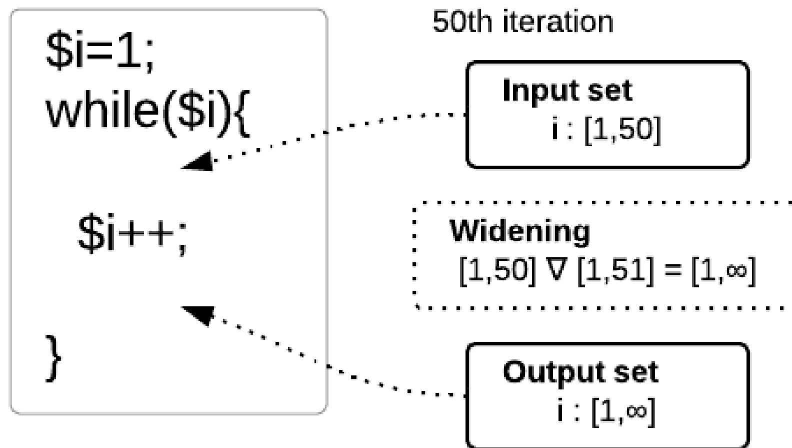
The facts stored in the input and output sets depend on the purpose of the particular analysis. However in general there are stored possible values and flags for program variables. There are several options for storing these values. First there can be stored particular values as they would appear in the program. Drawback of this solution is in possible memory consumption. Representing values of iterator variable within cycles could be very inefficient.

The other approach is in using abstraction domain. Values of a variable could be for example represented as intervals. This effectively solves the issue with memory consumption. An example of using interval abstraction is shown below.



As it can be seen from the figure above, it is sufficient to remember only single interval for the variable instead of many particular values. Of course using abstraction can cause loss of precision. It is important to find the right balance between memory consumption and needed precision according to particular kind of usage.

The next problem that can be seen in the figure above lies in number of iterations that would be needed for reaching fixpoint. If we omit value overflows the computation could never reach the fixpoint. For this purposes there can be defined widening operator that can predict behavior of variable values.



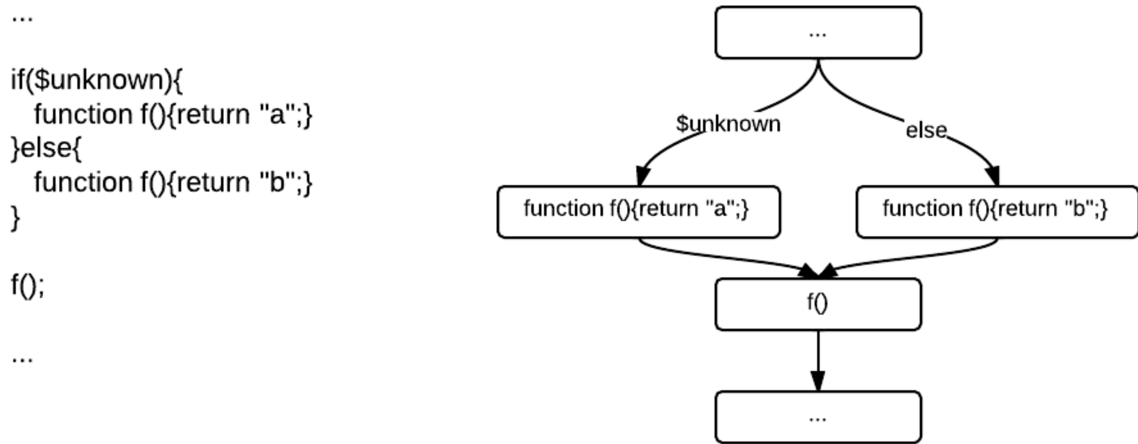
In the figure above can be seen that widening can speedup fixpoint convergence rapidly. However another loss of precision can be caused by using widening. Therefore analysis usually tries to use widening only after some limiting number of iterations until it tries to compute fixpoint precisely.

3.1.6 Static analysis of PHP

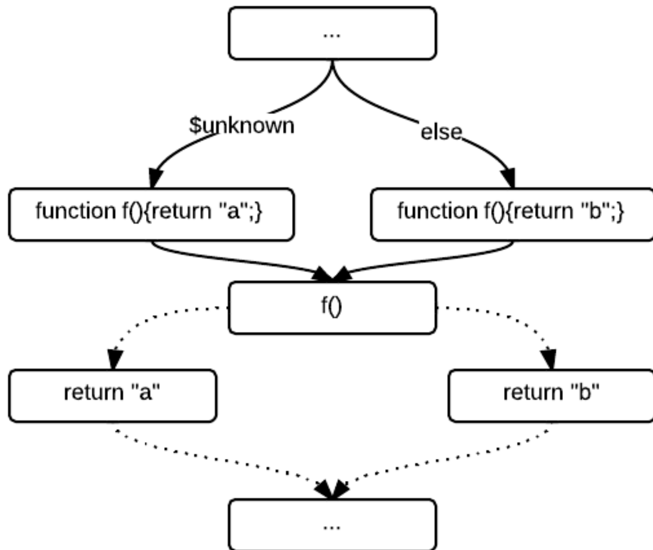
The static analysis that has been described above is sufficient for static strongly typed languages like C# or Java. The problem with dynamic languages lies in construction of control flow graph. In languages like PHP it is not possible to build intra-procedural control flow graph because of function are declared at runtime. There are problems with include statements that prevents even from building complete inter-procedural control flow graph.

Another dynamic feature of PHP is eval, that behaves similar to include, however the inserted source code is determined by value of its argument which can be variable or expression. Handling exceptions when analyzing dynamic languages is also difficult because we cannot precompute possible program flow paths caused by throw statements.

Handling dynamic control flow is the reason why dynamic control flow graph is needed. This structure is referred to as program point graph in our Weverca implementation. In the following examples we will show how the described dynamic constructs can be analyzed by our program point graph.



In the figure above we can see function declaration based on the value of the variable `$unknown`. This value cannot be known when building intra-procedural CFG, therefore it is not possible to predict which declaration should be used for the call `f()`. On the other hand the value can be known at time of analysis. This allows to remember possible declarations for `function f()` and use them for adding dynamic edges into the CFG as it is shown below.

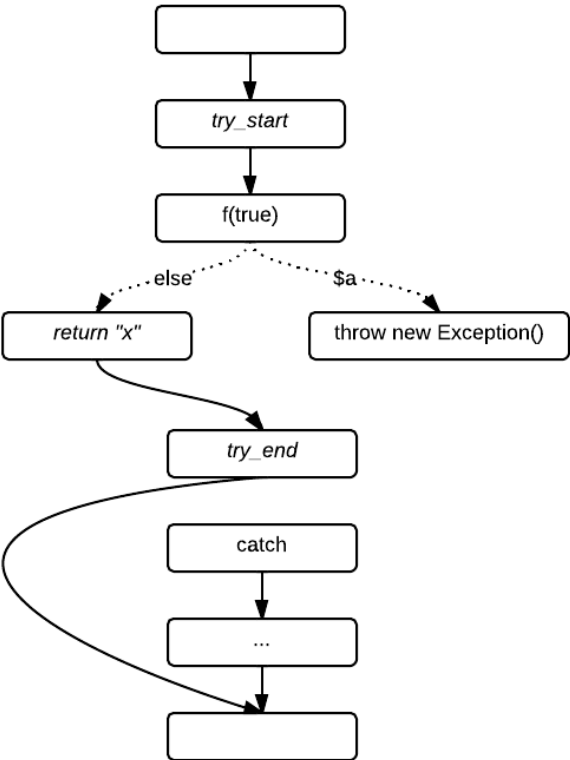


However situation can still occur during the analysis, when multiple possible function declarations for a call are available. This situation is shown at the figure above. Instead of using certain function declaration analysis has to consider multiple possible function declarations for the single call.

Other dynamic constructs that behave similarly as calls are includes and evals. These constructs can also be handled by adding dynamic edges into the CFG. A different approach is needed for handling exceptions. In static languages it is possible to connect throw statements with corresponding catch blocks at the time of building CFG. However function declaration are not known for calls in PHP while CFG is built. Therefore

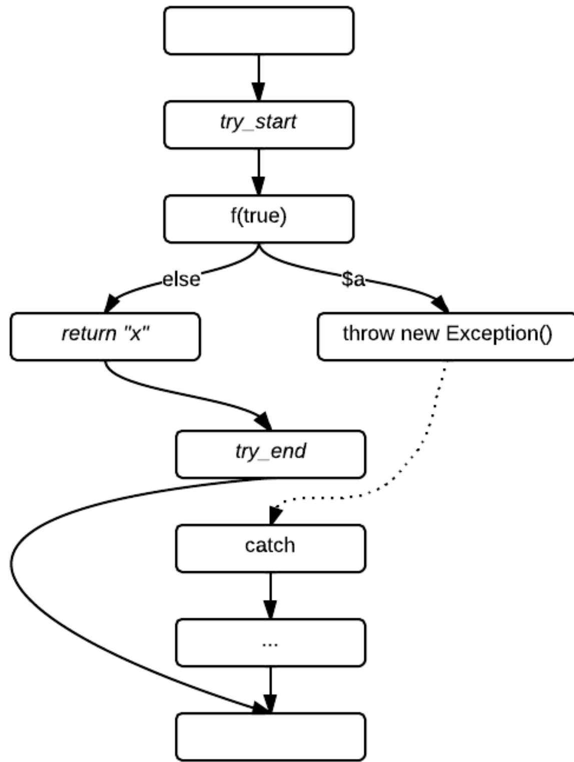
correspondence between throw and catch cannot be known in general. This situation is demonstrated by the following figure.

```
function f($a){
  if($a) throw new Exception();
  else return "x";
}

try{
  f(true);
}catch(Exception){
  ...
}
```



The above figure shows situation how analysis found matching function declaration for call `f(true)` and connects it dynamically to the cfg. However edges are not connected from throw statements to catch block because they can be dependent on conditional edges or another calls. Resolving of the edge from `throw` statement can be seen on the following figure.

```
function f($a){
  if($a) throw new Exception();
  else return "x";
}

try{
  f(true);
}catch(Exception){
  ...
}
```



When throw statement is analyzed it is possible to find encapsulating `try` block and accordingly find corresponding `catch` block. The CFG can be enhanced by adding the dynamic edge from the `throw` statement to the `catch` block.

3.2 Requirements

For compiling the source code we recommend to use Visual studio 2012 or newer. Project is compilable in .NET framework 4.5 or higher.

3.3 Architecture

Source code is parsed by Phalanger[4] parser component, which outputs abstract syntax tree (AST). AST is processed by metrics component, which computes metrics showing the quality of the source code.

AST is also used as an input of control-flow graph builder. This component outputs control-flow graph.

Analysis framework consists of following components:

- Program point graph builder
- Fixpoint computing
- Second phase fixpoint computing

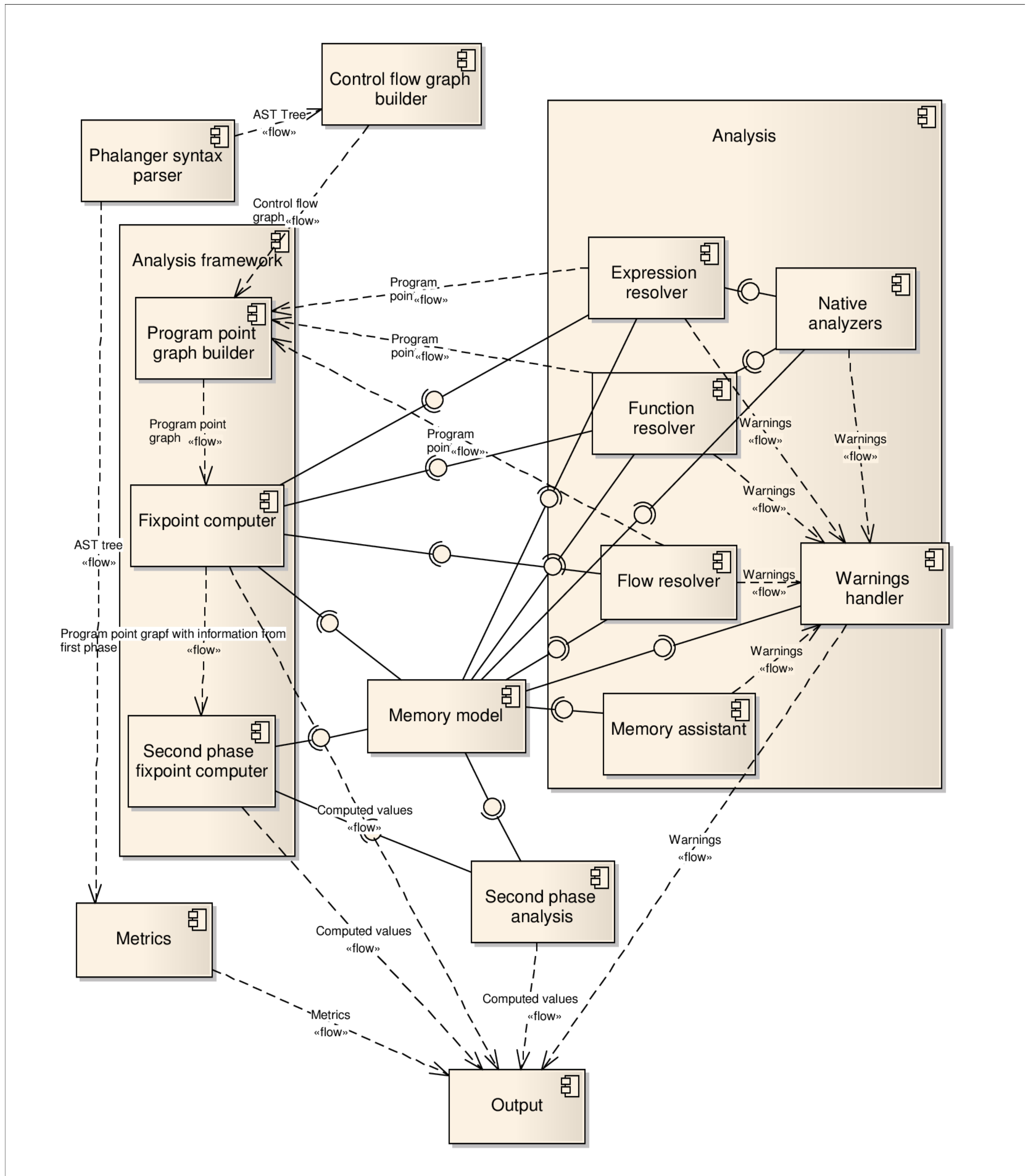
Program point graph builder processes given control-flow graph and outputs program point graph, which is used in component Fixpoint computer for fixpoint computation.

This component uses memory model for storing information about variables. During fixpoint algorithm following components are used for analysis of current program point:

- Expression resolver
    - evaluates arithmetic, logic and other expressions
    - handles object initialization
    - provides functionality for storing and reading static variables and constants
    - provides functionality for declaring classes
- Function resolver
    - resolve function calls and method, adds functions program points into existing program point graph
    - inserts information about function arguments and other variables into memory model, while initializing function call
- Flow resolver
    - provides functionality for exception analyzing
    - provides functionality for including files and resolving evals
    - handles condition evaluation and directs flow in fixpoint algorithm

Memory assistant provides functionality for memory model for actions which can produces analysis warnings. Native analyzers provides information about library functions, constants, and classes, which are defined as a part of PHP. Warning handler stores warnings produced by other analysis components into memory model.

Second phase fixpoint computer takes as input program point graph with computed values from the first phase, and runs second phase analysis. Second phase analysis is not implemented in the analyzer, but it is supported by framework. Second phase can tell e.g. which variables are tainted or which variables can influence the value of given variable in given program point.



3.4 Connection to Phalanger

Weverca uses syntax parser from Phalanger version 3.0 [4]. This version supports PHP 5.1 and also supports some features of PHP 5.3. Phalanger syntax parser parses source code and outputs abstract syntax tree.

Some changes had to be made in the Phalanger source code, because some important members of AST nodes were not public. These members couldn't be accessed from Weverca source code.
List of changes:

- *LabelStmt* - *VariableName Name* was made public
- *BinaryEx*
    - *Expression LeftExpr* - was made public
    - *Expression RightExpr*- was made public
    - *Operations PublicOperation* was added to allow access to *operation*
- *UnaryEx* - *Operations PublicOperation* was added to allow access to *operation*
- *IndirectFcnCall* - *Expression PublicNameExpr* was added to allow access to *NameExpr*
- *ActualParam* - *bool PublicAmpersand* was added to allow access to *ampersand*
- *ValueAssignEx* - *Operations PublicOperation* was added to allow access to *operation*
- *StaticMtdCall* - *TypeRef PublicTypeRef* was added to allow access to *typeRef*
- *StaticFieldUse* - *TypeRef TypeRef* was added to allow access to *typeRef*

3.5 Control-flow graph

Control-flow graph is a representation of source code using oriented combinatorial graph. Every node is called basic block and contains sequential pieces of code without any jumps or jump targets. Directed edges in control-flow graph represents jump statements.

In this project basic block is represented by class BasicBlock, which contains list of sequential AST nodes. Basic blocks are connected by different type of edges, all of them implements interface *IBasicBlockEdge*. There are three different types of edges:

1. *ConditionalEdge* - directed edge with condition. Analysis uses this edge only if the condition can be satisfied.
2. *DirectEdge* - directed edge without condition, analysis uses this edge if at least one of Condition edges can be false. The semantics of the edge is similar to else in if statement or to default in switch statement.
3. *ForEachSpecialEdge* - since control-flow graph doesn't have enough information about variables, it cannot create condition which will hold foreach iteration. That's why this special edge was created to give analysis information about foreach iteration.

Control-flow graph doesn't resolve function, method calls and includes. These statements are treated like sequential statements. Function, method and class declarations are also only copied into current basic block. Control flow graphs for functions and method are built on demand from analysis.

Special constructs in control-flow graph:

1) foreach statement - this construct is processed similarly to for cycle. Instead of a conditional edge which goes from end of cycle body to start of cycle body, we used *ForEachSpecialEdge*. Whole foreach AST node is stored into control-flow graph.
2) try and catch blocks - to resolve try and catch constructs we added new types of basic block:
    a) *TryBasicBlock* - represents classic basic block which starts with try statement. Stores additional information about associated catch block with current try block
    b) *CatchBasicBlock* - represents basic block, which starts with catch statement and contains additional information about catched exception

    In every basic block holds information about try block which ends in this basic block.

Control-flow graph cannot be build when:

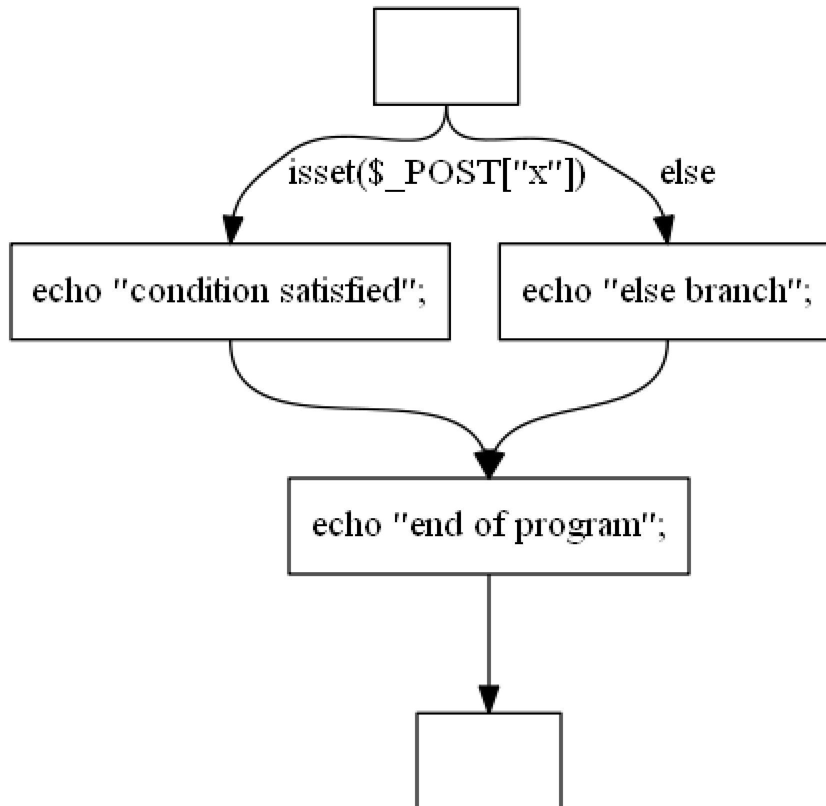
1. break is not in cycle
2. continue is not in cycle
3. target of goto doesn't exists
4. label is declared more than once

In all of this cases control-flow graph builder throws *ControlFlowException* and the analysis cannot start.

Control-flow graph is built in AST visitor (class *CFGVisitor*). For more details about building see generated documentation.

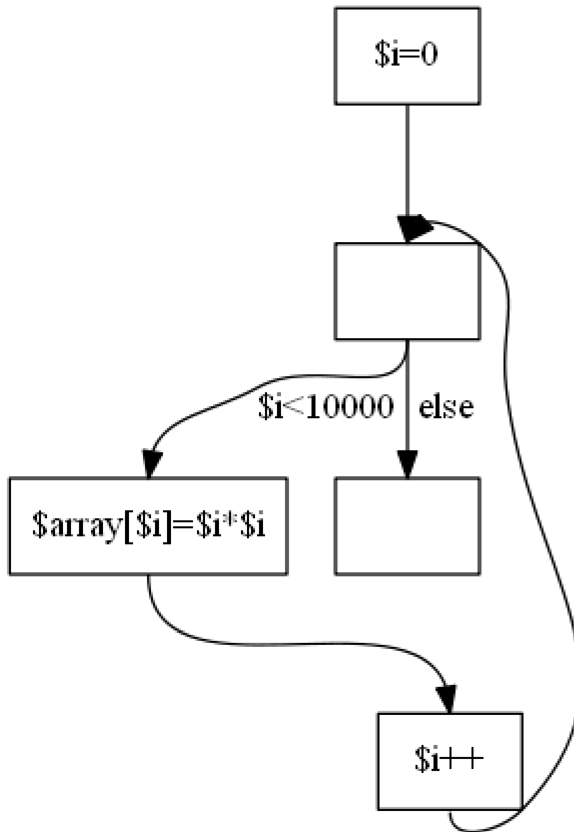
3.5.1 Examples

3.5.1.1 Condition example



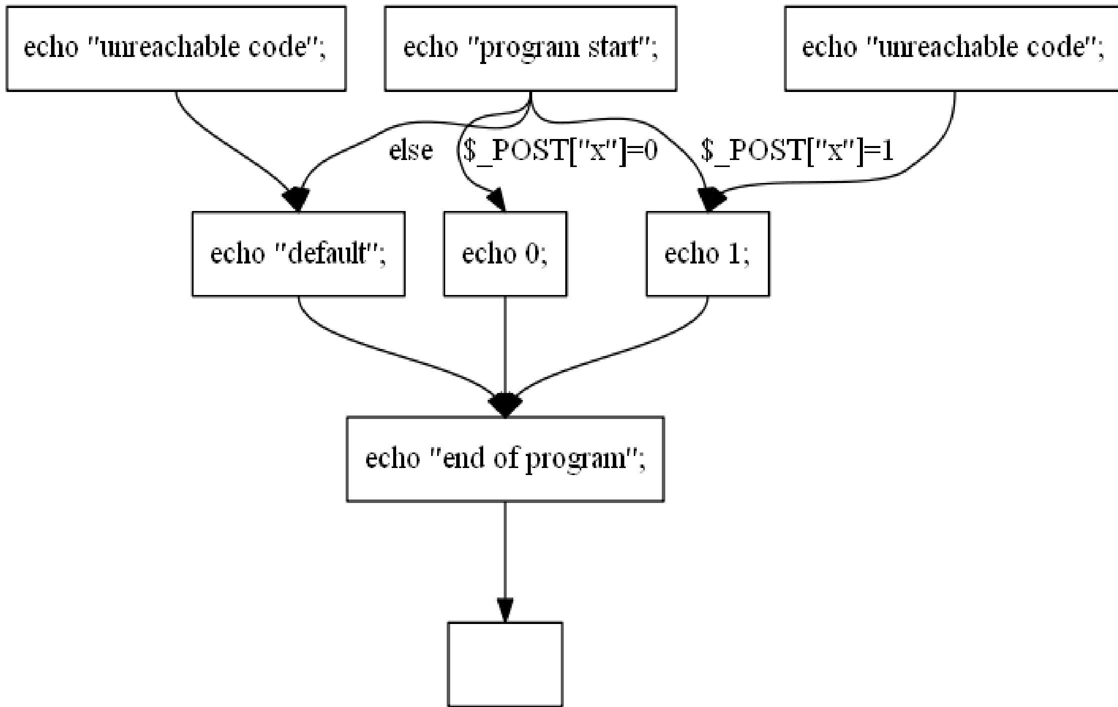

```
if(isset($_POST["x"]))
{
      echo "condition satisfied";
}
else
{
      echo "else branch";
}
echo "end of program";
```

3.5.1.2 Cycle example



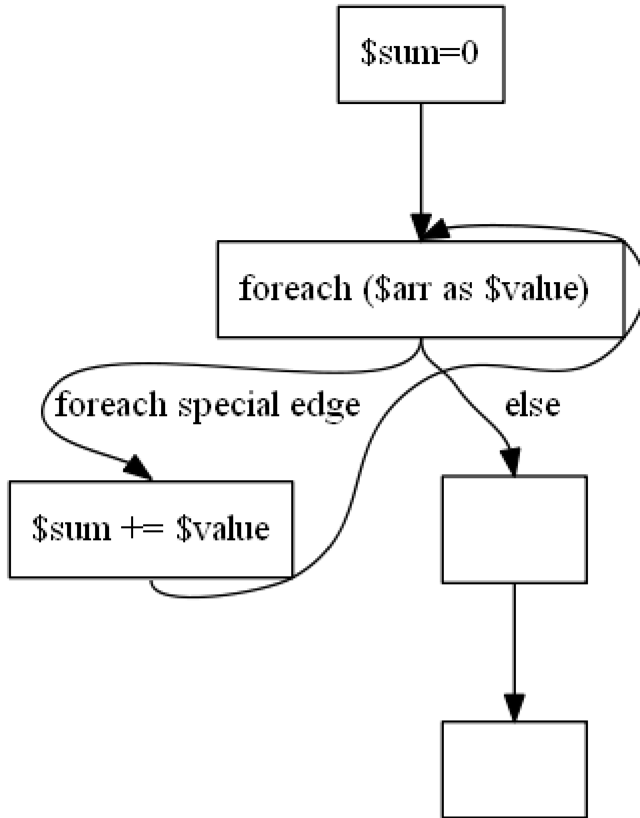

```
for($i=0;$i<10000;$i++)
{
$array[$i]=$i*$i;
}
```

3.5.1.3 Switch example



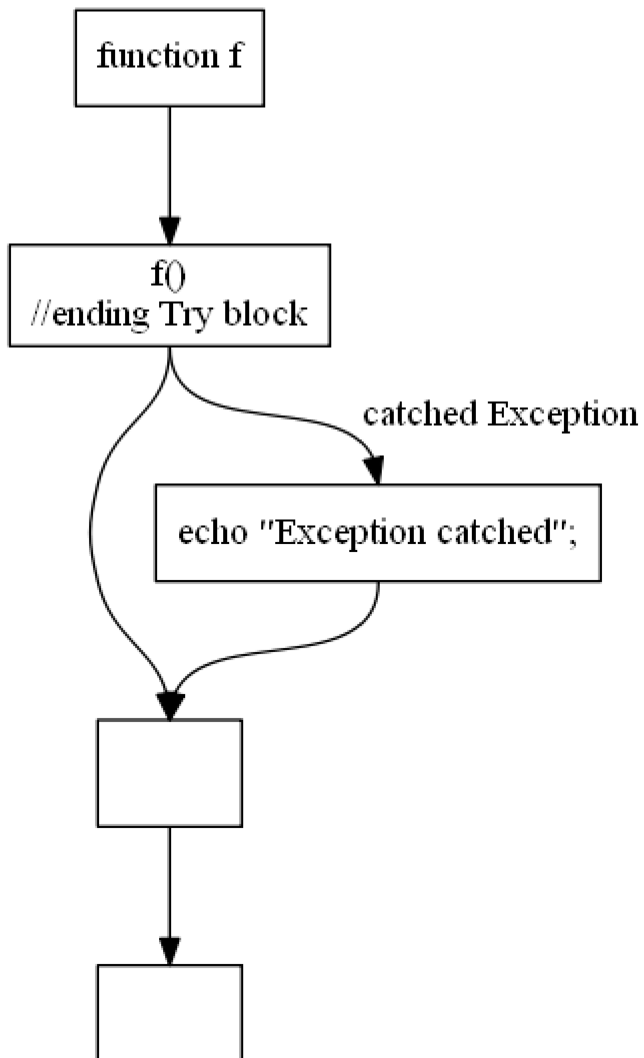

```
echo "program start";
switch($_POST["x"])
{
	case 0:
		echo 0;
	break;
	echo "unreachable code";
	case 1:
		echo 1;
	break;
	echo "unreachable code";
	default:
		echo "default";

}
echo "end of program";
```

For each example




```
$sum=0;
foreach ($arr as $value) {
    $sum += $value;
}
```

3.5.1.4 Exception in function example




```
function f()
{
      throw new Exception();
}
try
{
f();
}
catch(Exception $e)
{
echo "Exception catched";
}
```

3.6 Metrics

Software metric is a kind of measurement that investigates some property or characteristic of a piece of code. The measurement is important to assess different qualities of software that may be used for cost estimation, code optimization or just basic overall information about the entire product. In case of PHP analysis, we are mainly interested in characteristics that helps to debug a source code with security risks.

Metrics are evaluated statically in principle. The result is one simple information that can gives a hint about some aspects of the software. The advantage is that they give a rough information very fast. On the other hand, the result is approximate and imprecise, because the metrics try to reduce a complex program into simplified information. The meaning of each metric must be properly described, because it could be easily misunderstood.

Weverca provides framework for computing metrics of PHP source codes, that allows to evaluate both build-in and user/programmer implemented metrics. The framework gains source code data from Phalanger in the form of abstract syntax tree (AST). It has no access to control flow graph (CFG) that static analysis generates. This limits the strength of evaluation, but implementation of new metrics is easier.

The main idea of this framework is to compute code metrics for a single file and be able to merge these metrics for multiple files. The core of framework are *ProcessingServices* and *ProcessingService* classes. They can detect all implemented metrics via .NET reflection, process them and generate results. All results are gathered in *MetricInfo* class that is accessible from outside. This object contains a result value for each evaluated property and their occurrences in source code as nodes of AST, if they are logically defined for the specified metric. The framework distinguishes between three categories of metrics:

- Indicator metrics - They are used for checking presence of some measured quality in source files.
- Quantity metrics - These metrics measure number of occurrences of some quality in source files.
- Rating metrics - This category refers to metrics measuring score of some quality in source files.

Some metrics of each category are already implemented in Weverca. Business logic of metrics is inside classes that are derived from *MetricProcessor* class, implementing measuring functions. Each class must be annotated by *MetricAttribute* and implement both processing and merging methods. The annotation tells the framework which kind of qualities can be measured. The merging method solves the problem how to understand metric from more than one piece of source code. It is different for every category, thus there are three pre-implemented abstract classes derived from *MetricProcessor*: *IndicatorProcessor*, *QuantityProcessor* and *RatingProcessor*.

Since some properties are important for every programming language (e.g. number of lines), other properties are specific only for PHP or dynamic languages at

most (e.g. Magic methods or duck typing respectively). All of currently implemented metrics are listed in next chapters.

3.6.1 User-implemented metrics

Weverca gives a possibility to create custom metrics to programmers. Programmer must add record into one of *ConstructIndicator*, *Rating* or *Quantity* enumerations according to the type of the metric. Implemented class must be derived from one of *IndicatorProcessor*, *QualityProcessor* or *RatingProcessor*. Merging method can be omitted if implemented one is sufficient. It is also necessary to mark the class with *MetricAttribute* that takes the enumeration value as its parameter.

The usual way to compute metric in process method is to traverse AST. The method gets Phalanger syntax parser as parameter. We can visit every node of AST, because Phalanger provides *TreeVisitor* visitor class. We create a class inherited from it and method for desirable AST nodes can be overridden. The *SyntaxParser* provides even more information about source code.

3.6.2 Class presence

Class presence metric determines whether there is at least one class declared in the given code. The metric works over the list of types used in the code provided by *SyntaxParser*.

The result of the metric is *true*, if there is a class or *false* if there is none. The metric also returns the list of all declared classes.

3.6.3 MySQL or other SQL functions usage

This metric checks if there is any of MySQL functions used in the code. It goes thru the AST and checks the names of called functions. The result of the metric is *true* if any of the the functions called in the code is on the list of MySQL functions or *false* if there is none of the functions used. The metric also returns the list of occurrences of these calls.

3.6.4 Dereference with double $

This metric checks if there is dynamic dereference of a variable present in the code. It works in the same ways as MySQL presence metric, but while going through the AST the presence of different construct is being checked.

The result of the metric is *true* if dynamic dereference of a variable is used. Otherwise the result is *false*. The metric also returns the list of occurrences of the calls.

3.6.5 Maximal depth of method overriding

This metric first creates the trees of inheritance using the list of types provided by *SyntaxParser*. Then the trees are being processed to find the maximum distance between method declaration and its farthest overriding. The distance is calculated for each branch

of the tree. The maximum of these distances is the result of this metric. It also returns the occurrences of the method with maximum depth of overriding.

3.6.6 Maximum inheritance depth

The metric gets all types in script provided by *SyntaxParser* and for every class, it traverse all its ancestor. It returns all classes of the longest chain of inheritance from the most derived class to the class without ancestor and its length as the quantity value of the metric.

3.6.7 Dynamic calls and object creations

Dynamic call is similar to the dynamic dereference, but in this case the dereferenced value is not used as a name of a variable, but as a name of a method or class. The metric itself works the same as the metric for dynamic dereference, but the AST is checked for a class and object creations and indirect method calls instead of variable usage.

3.6.8 Class alias

This metric check if there is an alias created in the code. It works in the same way as the MySQL metric. But instead of checking a presence of one of MySQL methods, a use of *class_alias* is checked.

The result of the metric is *true*, if a creation of class alias is used. Otherwise the result is *false*. The metric also return the occurrences of alias creations.

3.6.9 Magic methods

The metric traverse all types in source code provided by *SyntaxParser* and finds all methods that has name of a magic method starting with double underscore.

3.6.10 Dynamic inclusion

We identify, if parameter of the `include` or `include_once` statement can be evaluated in compile-time, so if the expression contains only literals and concatenation. If not, it must be evaluated dynamically. The metric returns all these inclusions.

3.6.11 Number of lines and source files

These metrics are only simple statistics. They show that the Phalanger parser stores various information in the AST.

3.6.12 Superglobals

Super global variable are built-in variables always defined in every script, e.g. `$_GET`, `$_POST` or `$_ENV`. We traverse the entire AST for occurrences of these variables usage and return them as AST nodes.

3.6.13 Function/type declaration inside function body

The metric traverses all subroutines in the script and returns all definitions of a function or type.

3.6.14 Duck typing

The metric returns all accesses to object by a member, because every such access may be regarded as duck typing.

3.6.15 Passing variable by reference at call side

At first, we check all subroutines and collect all such that have at least one parameter passed by reference. The metric returns all calls of these subroutines.

3.6.16 Autoload

The metric finds declaration of `__autoload` function and also all declarations of function or method, that occurred as parameter in `spl_autoload_register` function and so it can be registered as new autoload function.

3.6.17 Class coupling

The metric finds all class definitions. Then finds all couplings between classes, i.e. for every class, all occurrences of other classes inside it. The result rating is calculated as ratio between sum of all occurrences and number of classes. This is static information acquired from static references that can appear when there is object creation or static method call.

3.6.18 Functions coupling

This is similar to class coupling. The metric finds all function definitions and then all couplings between these functions, i.e. for every function, all other function calls (not recursion) inside its body. The result rating is calculated as ratio between sum of all function calls and number of functions. Methods are taken as part of classes.

3.6.19 Eval

The metric simple returns all occurrences of `eval` call by traversing entire AST of source code.

3.6.20 Session functions

The metric simple returns all occurrences of all session function calls by traversing entire AST of source code.

3.6.21 References

This metric checks if there is use of references in the code by going through the AST. The result of the metric is *true*, if creation of some kind of references is used. Otherwise the result is *false*. The metric also return the occurrences of reference creations.

3.7 Framework for analysis

3.7.1 Forward analysis

The main purpose of Weverca project is to provide ability to run forward static analysis on given Control-flow graph. This is covered by *ForwardAnalysisBase* class, which is the entry point class of the framework.

Before the class can be used for analysis it is needed to implement *ForwardAnalysisBase's* abstract methods. These methods determine which memory and computational model will be used for analysis specified by creating appropriate resolvers and snapshots.

After particular specialization of the analysis is created, input environment for the script can be initialized through *EntryInput* member of the analysis object. This input is used as input set of program flow when analysis is started by calling *Analyze* method.

Analysis can be also influenced by parameters changing analysis precision and convertibility. Namely these parameters are *WideningLimit* and *SimplifyLimit*. *WideningLimit* tells the analysis, how many times it could process each program point before widening is applied. This allow to speed up analysis of too complicated program parts by using over approximation. *SimplifyLimit* then limits the number of possible values stored within single *MemoryEntry*. It also speeds up some kind of complex situations at the expense of an analysis precision.

3.7.2 Program point

Basic unit of analysis is *ProgramPointBase*. There is program point for every supported Phalanger's abstract syntax element which is important for analysis. In addition there are special program points, that doesn't have counterpart between syntax elements.

Main three groups of non-special program points are value points, left value points and declaration points. Value points represents some expression or syntactic construct that produce some value. For example literal, function call, assign, etc. Left value can also produce some value, however it can in addition be assigned by a value. Examples of left values are variable or object field usage. Declaration points belongs to function or type declarations.

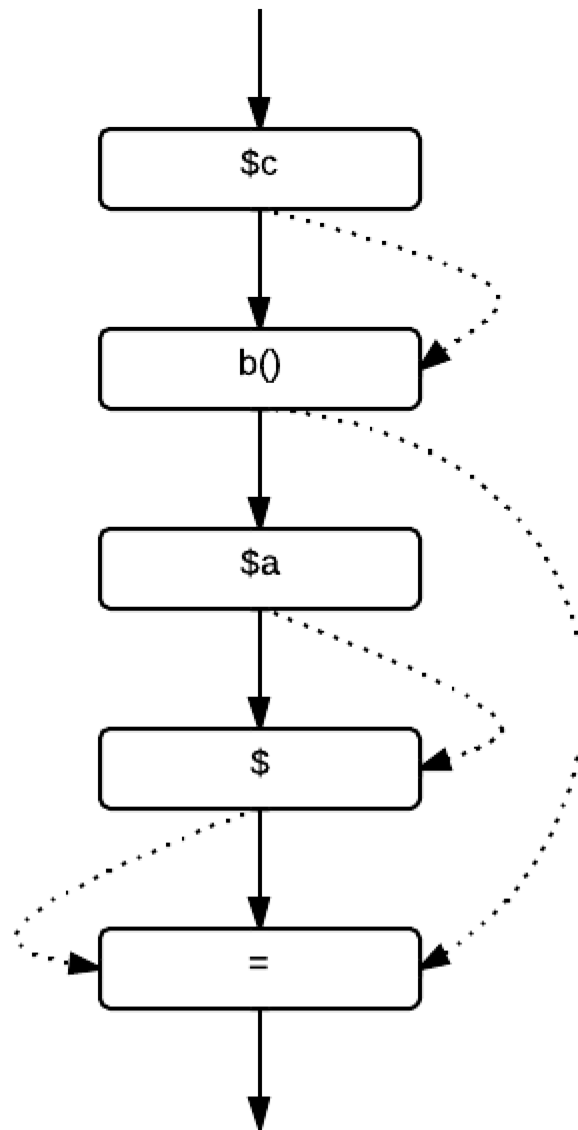
The last group of program points is called special points. There are points for describing some special semantic that is not present within the syntax tree. They are assumption points describing feasibility of states based on conditions. Also there are program points for defining boundaries of `try` and `catch` blocks. Lastly program points for support dynamic calls, includes and evals are present. All these special program points will be described in detail later.

3.7.3 Program point graph

With a single program point we are able to describe single syntactic or semantic part of an analyzed program. To describe whole program from the point of view of the Control-flow graph, connecting program points into program point graph is needed.

Program point graph is a structure consisting of program points as nodes and edges between them. There are two types of edges. Flow edges connecting program points in direction of program flow. These edges define ordering in which program statements can be executed. The other type of edges are value edges that reference operands from operators within expression. Thanks to these edges are operators able to work with computed values of their operands.

Building of program point graph is based on walking through Control-flow graph. Every statement of Control-flow graph is split into statement elements. From these elements program points connected in postfix order are created and connected into program point chains. The order is important because of ensuring that operand values are evaluated before they are needed by operator. Figure below shows example of program point chain created from statement `$$a=b($c)`.

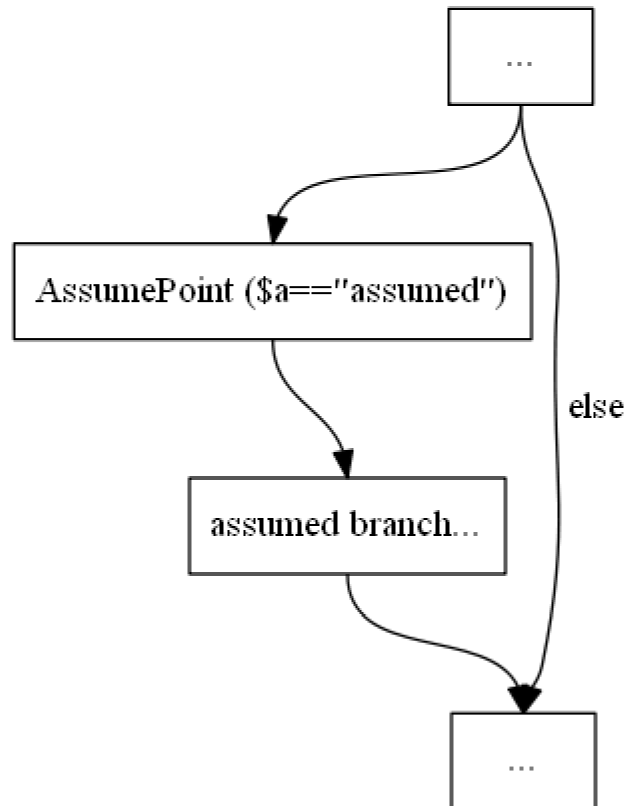


In the figure above there are flow edges shown as solid lines. These edges determine direction of program flow, which is in the statement same as postfix order. Dotted lines are used for value edges. These edges define operands for operators. Left value edge is connected to left side of program point representation, right value edges are connected at right side.

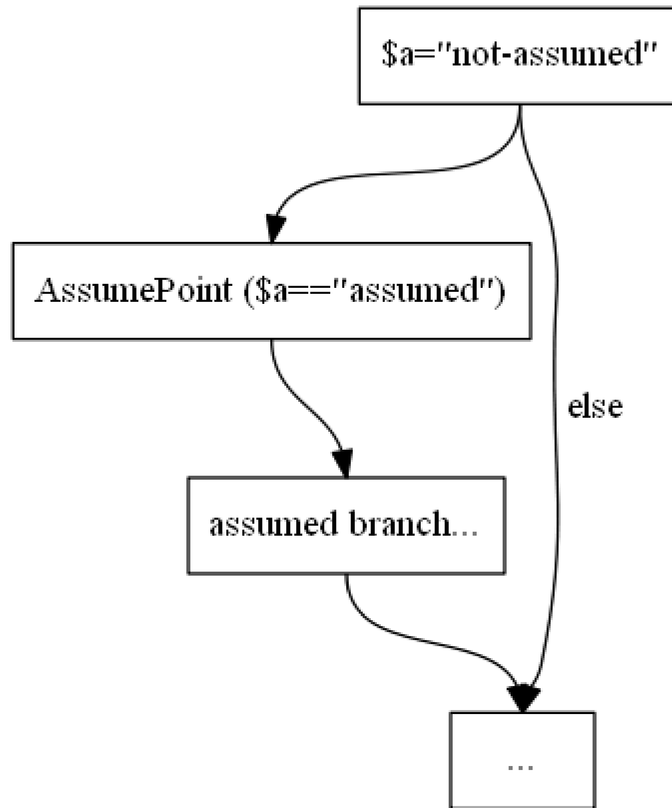
Program point chains are connected together according to edges in control flow graph. For statements connected with condition-less edges is sufficient to add flow edge connecting chains. However if condition on edge is present, *AssumePoint* between chains is needed. Value edges cannot been propagated through different program point chains, because it would mean that there is result of an expression directly shared between two different statements. But in PHP there is not such a construct.

3.7.4 Assumptions

Representing conditional statements in static analysis is quite different from usual conditions known from programming languages. Conditions are rather resolved as assumptions on environment state by *AssumePoint*.



In the figure above we can see that even if we don't know anything about value of `$unknown` variable it can be assumed that in the conditional branch of program flow has to be equal to "`assumed`". Of course in some situations we have better information about condition.



As can be seen above, there is no possible assumption in the current environment state. It means that assumed branch is not feasible and analysis does not need to evaluate it.

3.7.5 Try blocks

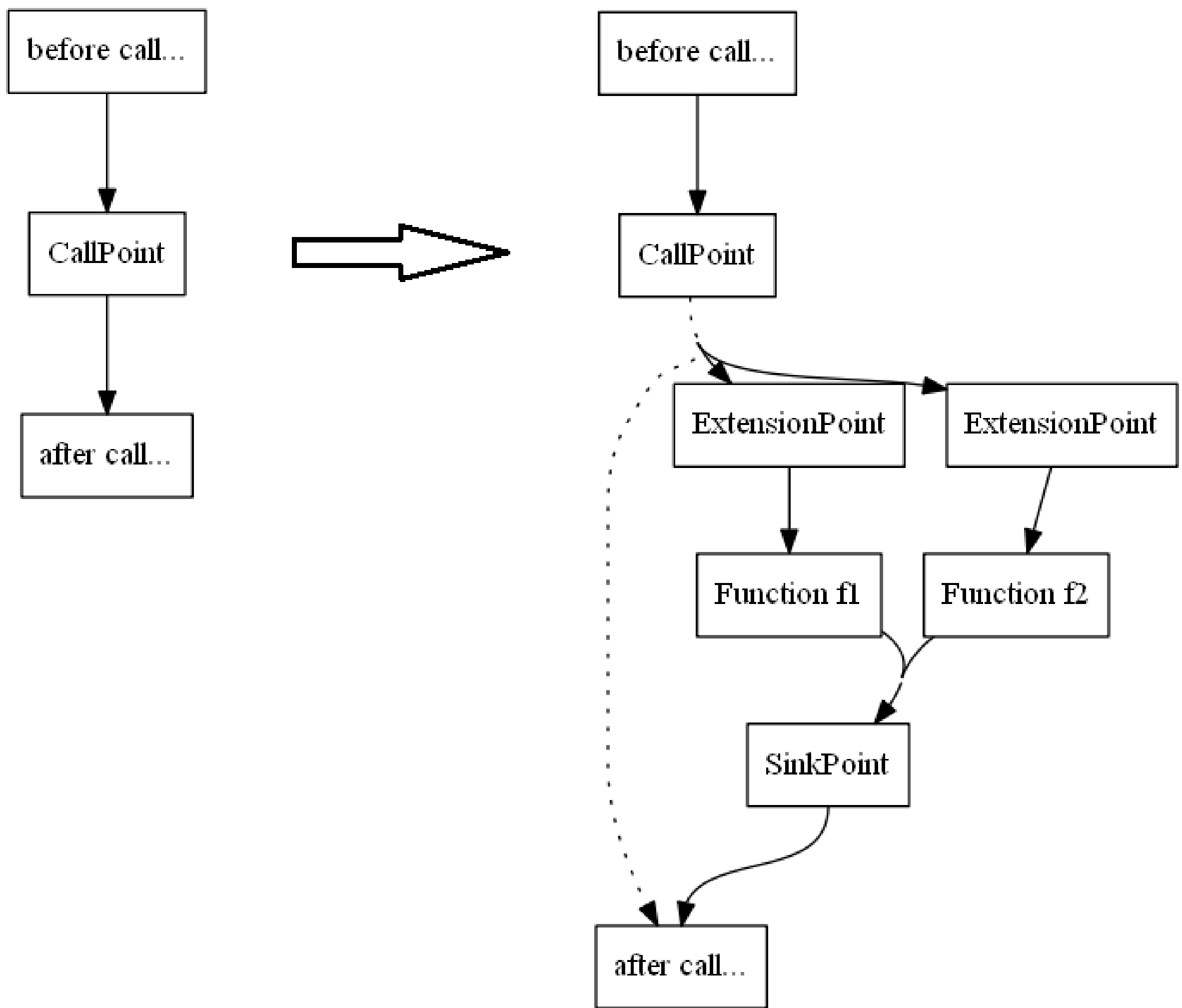
In input Control-flow graph are `try` blocks represented by collection of related catch blocks. Program point graph creates program points for `try` block beginning and ending. These program points mark the `try` block scope.

The scope boundaries keep information about `catch` blocks that belongs to `try` block. Also there is reference on program point sub-graphs created for every `catch` block. These `catch` blocks are connected during analysis according to thrown exceptions.

3.7.6 Extension branches

Program point graph as described above would be sufficient for languages without dynamic function calls, includes and evals. These constructs are not known at the time of building program point graph. For this reason ability to modify program point graph during fixpoint computation is needed.

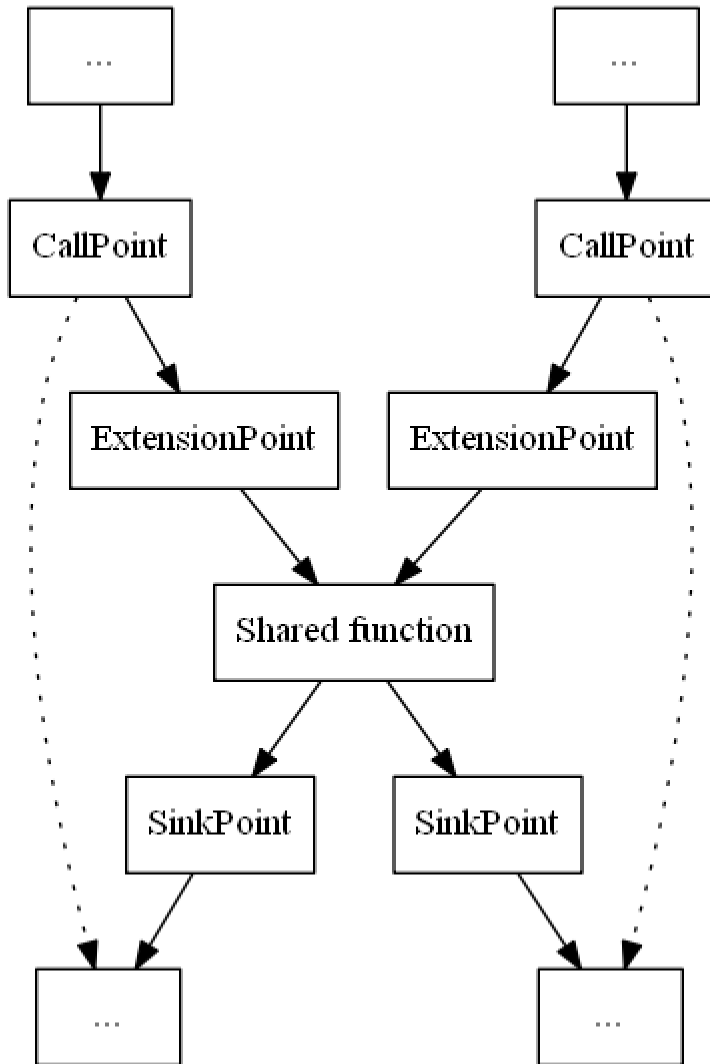
This ability is provided by *FlowExtension*, which can connect extension branches between any program point and its children. Following figure shows an example of extending a program point by two program point graphs of a function with two possible declarations.



Because of the possibility to have different initial environment for every extension branch there is *ExtensionPoint* prepending each branch. In this point can be set function arguments or other information defined by analysis. All extension branches are merged into *SinkPoint* which is created for every program point. Here return values and merging branch environments are processed. It should be noticed that return value from *SinkPoint* is forwarded to the owning value point. This mechanism solves value propagation from extensions into extended program points.

3.7.7 Sharing program point graphs

In some cases analysis needs to limit the number of created program point graphs. The typical situation is limiting the maximal depth of recursion. For this purpose the analysis framework provides possibility to share single program point graph between multiple flow extensions. In the following figure there is an example of function shared between two calls.



Sharing program point graphs prevents analysis from constructing unbounded program point graphs. However it has to be noticed that shared program point graphs reduce analyses precision by merging contexts from different parts of program point graph.

3.7.8 Fixpoint computation

Having all parameters and resolvers prepared, analysis can be started. Firstly it creates program point graph from entry Control-flow graph. Then input of the created Program point graph is initialized by *EntryInput*, because of the possibility to define initial environment of an analyzed program. After initialization is done fixpoint computation starts. Program points from program point graph are visited by *FlowThrough* method one by one. The order of visiting is defined by flow edges between points and by *WorkList* class that optimize computation of program points with many ancestors.

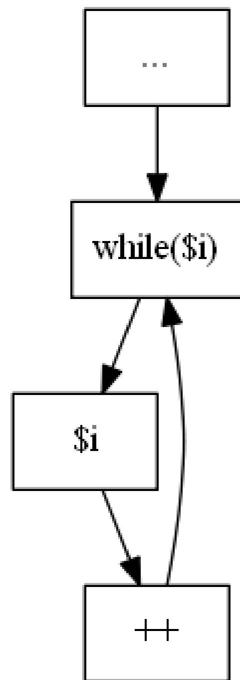
Flow through a program point means in terms of static analysis describing how the program point influences an environment of the analyzed program. There is state of the environment described by program point's flow input set. When *FlowThrough* is

completed, the flow output set will contain environment state after processing program instruction represented by the program point.

It has to be noticed that program point graph can contain cycles. Because of this fact changing of flow output set of any program point may result in changing its flow input set. That is the reason why more flow iterations for single program point may be needed. Every iteration adds some new facts about environment state of program point.

The goal of the fixpoint computation is reaching a state when flowing through any program point in the graph does not change its flow output set. Finding the state is done by worklist algorithm that keeps list of non-processed program points. In every step one of the points is taken and its *FlowThrough* is called. Then every program point which flow input set has been changed is added to the list. Fixpoint is found if there is no program point to be taken from the list.

Finding of described fixpoint can be very time and memory exhaustive. Simple example can be unbounded loop with iterating variable as shown below.



In every iteration there is one possible value for variable `$i` added into the flow input set. If the overflow handling is omitted the fixpoint cannot be ever found. It is needed to use widening in such cases. According to *WideningLimit* the framework will use widening to predict trend of flow set changes to speed up the computation. This approach offers balancing between computation time and accuracy that can be adjusted for every analyzed problem.

3.7.9 Second phase analysis

The main goal of forward analysis is to create memory structures and resolve includes, functions and other dynamic features of php. Even if we could collect any other

information related to the domain that we are analyzing, it could be advantageous to split analyzing process into two phases.

In first phase we collect runtime information about calls, includes and other operations changing a program point graph by using *ForwardAnalysisBase*. Result of the analysis is the program point graph with all needed extensions connected. All program points in the program point graph are also filled with computed fixpoint information.

This program point graph can be used by *NextPhaseAnalysis*, which operates on program point graphs without changing their structure. Because of the stable program point graph structure it is possible to process forward and backward analysis. This is useful for collecting meta-information like values flag propagation, type optimizations for compiler, etc.

To benefit from static memory structures in second phase of analysis, memory models supports two operational modes. In the first one which is called *MemoryLevel*, structures can be created and changed during analysis by assigning aliases, arrays or objects. In the second mode, that is called *InfoLevel*, the structures are static and only meta-information values can be propagated through them according to existing aliases. This ensures same behavior as meta-information would be collected by *ForwardAnalysisBase*, but with the advantage of separated computational models.

3.8 Memory models

Static analyzer needs to store information about memory state in each program point of control flow graph. This includes storing information about variables, objects, arrays, and aliases between variables, indices and object fields. It is necessary even to store values in statically unknown memory locations (e.g. statically unknown index of an array). In scripting languages without type check, it is also possible to have values of different types in a single variable. This can all happen in any program point.

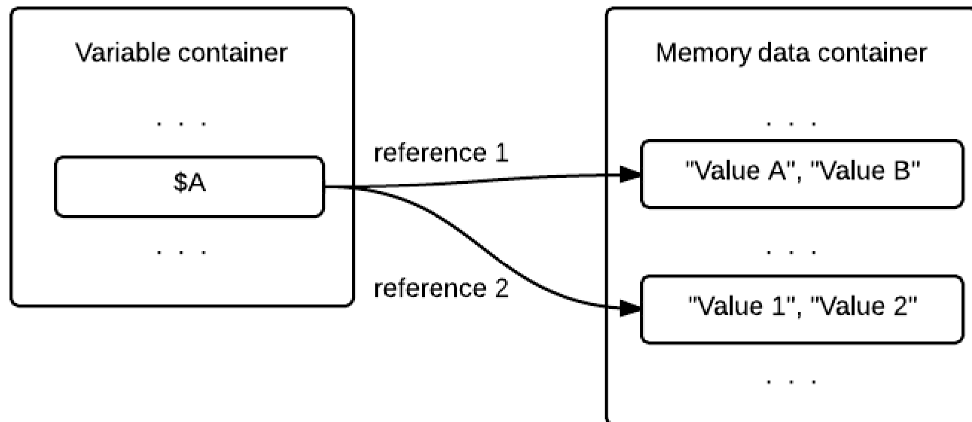
Memory model is analysis component which has been introduced in order to store all these informations in a single place.

Architecture of the analyzer was designed to allow programmers to create their own implementation of memory model. Programmer can easily start new analysis with memory model which best fit the problem analysis is meant for.

3.8.1 Virtual Reference memory model

Main purpose of the Virtual reference model implementation is to provide analysis framework users with lightweight memory model. This is done by memory abstraction, where data are stored in memory places, accessible through *VirtualReference*. Every *VirtualReference* can store single *MemoryEntry* within single *Snapshot*.

For description of variables, array indices and object fields, that will be called storages, there is a *VariableInfo* that contains set of virtual references. Schema of memory model abstraction is shown below.



On the figure there is a single storage for variable `$A` containing two virtual references. Each of virtual references identifies *MemoryEntry* with two possible values. There can be seen multiple levels where uncertainty comes into play.

3.8.1.1 Accessing memory

Reads and writes on memory in context of *Snapshot* are based on *MemoryEntry* association to virtual references belonging to read or written storage. Every storage, even those representing object fields or array indices, are stored in the same way as a simple variables. In almost all cases virtual reference provides key to accessed *MemoryEntry*. But there also exists special virtual references described later.

Sometimes it can happen, that analysis needs reading and writing of storage determined by several possible names. These operations are resolved in same way as single storage with references merged from storage references for every possible name.

More advanced case for memory access is accessing completely statically unknown storage, which is storage without any known approximation of possible names. Supporting of this feature will add significant overhead into virtual memory model implementation. Because of focusing on lightway implementation of virtual reference model there is lack of this feature for now. If supporting of unknown storage is needed, copy memory model has to be used.

Because of uncertainty of analysis it is possible that single storage can contain multiple virtual references. When trying to read such a storage, every *MemoryEntry* stored for contained virtual reference has to be merged together.

Writing into storage is more complicated. We distinguish following scenarios according to the number of stored virtual references.

- <u>no reference</u> - there is an attempt to write to non-allocated variable. Implicit reference is created and associated with storage. Simple write as in case of single reference is done.

- single reference - strong write is processed. It means that written *MemoryEntry* is associated with the reference.
- multiple references - weak update is applied. Written value is merged with *MemoryEntry* already associated with a reference. This is applied to all storage's references.

3.8.1.2 Aliasing

Main advantage of Virtual reference model is simple way to represent memory aliasing. If alias of one storage should be assigned into another storage, it is sufficient to copy virtual references of aliased storage to the aliasing one. This works similar as internals of php do.

The drawback of this solution is missing ability for Write-Read support caused by analysis uncertainty. It means that reading storage may produce another value from that was previously written. It may happen in cases when the storage contains multiple virtual references and has to process weak update.

3.8.1.3 Variable containers

Every Virtual reference model's snapshot contain multiple storages for variables. Those storages keep information which variables are defined within context of the snapshot. There is a list of used variable containers

- Local variables - Here are stored variables declared within local context of call
- Global variables - Storage of variables declared in global context
- Local control variables - Control variables used by analysis within local context of call
- Global control variables - Control variables used by analysis within global context
- Meta variables - Variables used for storing meta-information needed for internal purposes of memory model. Here are stored for example values of object fields, object types, etc.

From the list above can be seen that global and local variables are stored in different containers. However sometimes it is needed to have global variable accessible through local alias. This is done by simple aliasing of global variable from the local one. It results in having virtual references of global variables within storage for local variables.

3.8.1.4 Storing structured values

Virtual reference model distinguish two types of structured values. They are object values and array values. Both types of structured values behave similarly in the way of storing nested values. These nested values are represented by special variable stored in Meta variables container. Variable name consists of identifier of an index or a field and unique id of its parent (value storing given member). This ensures unique name which can be used for accessing member in context of snapshot.

Described representation benefits from correct semantic with keeping same storage behavior as with usual variables. It means that same algorithm for merging, values reading and writing can be used for both structured and scalar values. This also

works with storing multidimensional arrays, where next dimension array is just the array stored at index storage of array with preceding dimension.

However there is one exception needed for working with arrays. It is needed because of php array copy semantics. If array is assigned into a storage, it has to be copied, because of avoiding of incorrect value propagation.

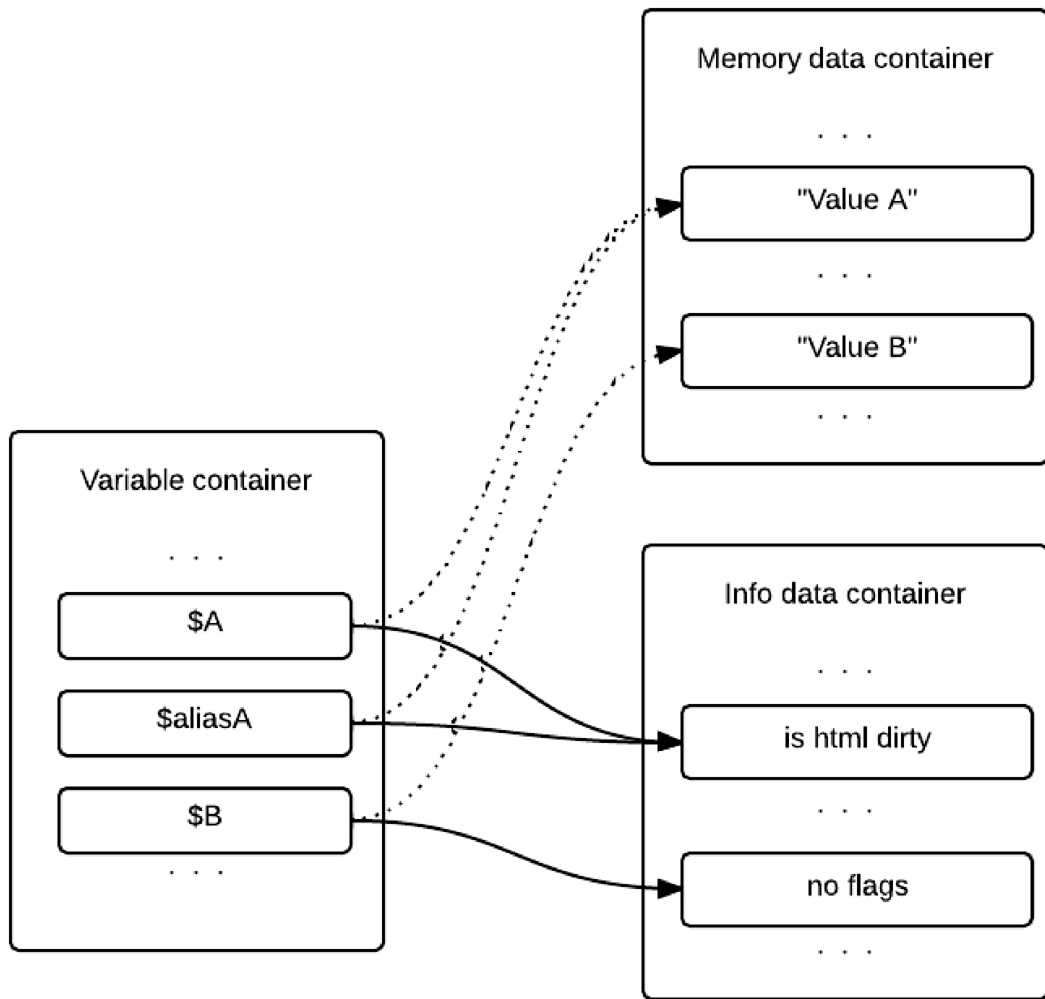
3.8.1.5 Special references

In most cases Virtual reference memory model uses references for identifying particular *MemoryEntry* within data container. However there are some situations where value of reference has to be determined by analysis. It may happen when attempting to read index of non-array value, obtaining field of non-objects etc. In these cases appropriate handler through *MemoryAssistant* has to be called to define result values or create warning logs.

These cases are solved via lazy evaluated virtual references that specify callbacks for read and write attempts. Then if reference value is needed, read callback is invoked to provide assisted value. Same behaviour is used for writing with write callback.

3.8.1.6 Second phase analysis support

Because of semantic of virtual references that define memory structure it is easy to provide support for *InfoLevel* operational mode. Everything that should be done is switching memory data container to info data container. While keeping previously created virtual references it is possible to propagate info values in the same way as in *MemoryLevel*. How does it work is shown on the figure below.



In the figure there is a snapshot where writing info value into `$aliasA` results in ability to read same info value from `$A` variable because their references identifies same *MemoryEntry*. The `$B` variable of course remains unchanged. This is necessarily for analyses working with flag propagation.

3.8.2 Copy memory model

Copy memory model represents implementation of memory model which tries to ensure write-read semantics for every memory location. Write-read semantics means that when you write a value to some memory location you will read exactly the same value on next read.

Ensuring write-read semantics consumes more time and space. Virtual Reference memory model should be used when time complexity is more crucial than precision of result of analysis.

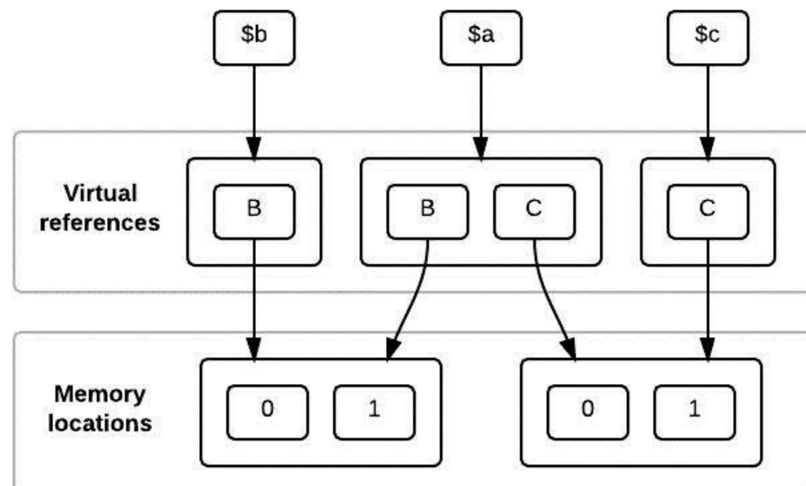
3.8.2.1 Difference between Copy and Virtual Reference memory models

The main difference between Copy and Virtual Reference memory models is that Virtual Reference model saves values of aliased memory locations on many different memory entries which are connected by virtual references. This approach is less complex but an existence of uncertain may alias can corrupt write-read semantics.

Code below illustrates corrupted write-read semantics:

```
$a = 0; $b = 0; $c = 0;

if ($_POST[?]) $a = &$b;
else $a = &$c;

$a = 1;
```

Program contains uncertain (MAY) aliases between variables. When you run this code there will be two possible values for variables B and C. Variable A will contain just single value no matter the visited branch of condition.

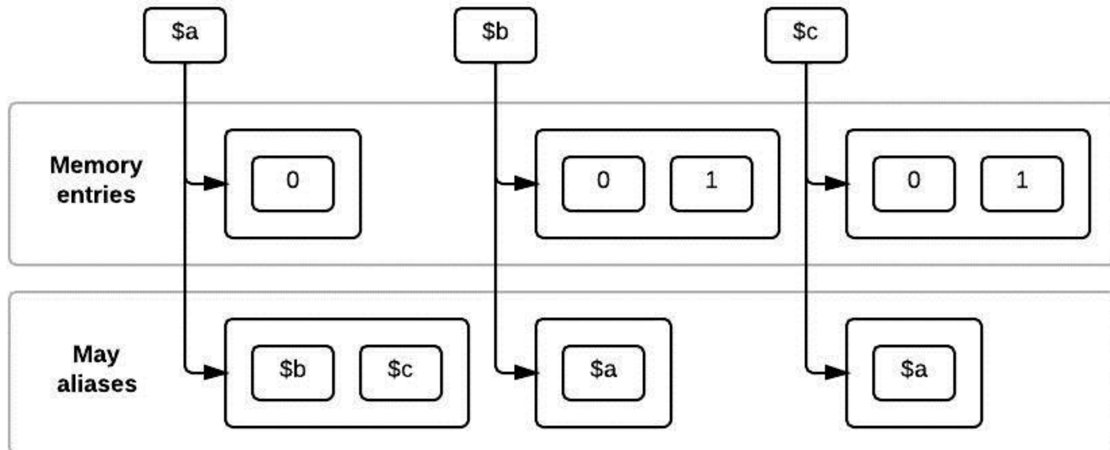


-Memory representation in Virtual References memory model -

Virtual reference model weakly updates two memory locations for two uncertain aliases. When the value of A is requested, both memory locations are collected and result

of analysis is that there can be both values in variable A. On the next diagram there is final memory snapshot of Virtual References memory model.

Copy memory model uses strong read-write semantics. Every memory location contains copy of all data which can be found in it. The data are always replicated even when there is alias link between two or more memory locations. Information about alias links is separated from the data itself. Using copy memory model analysis can strongly update memory location of variable A and weakly update B and C. Next diagram shows memory snapshot at the end of program:



-Memory representation in Copy memory model -

Implementation of write-read semantics increases precision of analysis. Disadvantage is that updating of memory snapshot is more time complex and consumes more space. Virtual Reference memory model should be used when time complexity is more crucial than precision of result of analysis.

3.8.2.2 Variable access paths

Copy memory model identifies every memory location using its access path. The access path corresponds to PHP constructs for accessing variables, arrays and object fields:

```
$variable
$variable['index’]
$variable['level1']['level2']...
$variable[$variable2]
$variable[$variable2['index']]...
$variable->field
$variable->field[‘index’]
$variable->field1[‘index’]->field2...
$variable->$field
```

The access path defines each of memory locations (class *MemoryIndex*). The access path provides indexing between memory locations and data entries within the single snapshot. MemoryIndex is also used to point between memory locations when there is some connection between them (aliases).

3.8.2.3 Difference between arrays and objects

It would be nice if objects and arrays had the same memory semantics in PHP. Sadly, the semantics of array and objects is different - copy semantics for arrays vs. reference semantics for objects.

Consider this PHP code:

```
$arr = array();
$obj = new object();
$arr2 = $arr;
$obj2 = $obj;

$arr2[1] = 1;
$obj2->a = 1;
```

Result is completely different because of difference between copy and reference semantics for objects and arrays. `$arr` and `$arr2` contain two different arrays in contrast to $obj and $obj2 with shared reference to single object. So update of each array is completely independent but update of object has to modify two different memory locations.

If it is necessary, reference semantics can be forced even for arrays by creating an alias between two variables in PHP:

```
$arr = array();
$arr2 = & $arr;
$arr2[1] = 1;
```

Modification of $arr2 now changes two different memory locations.

Memory model has to model this PHP behavior and provide semantically correct reading and writing to certain and uncertain memory locations.

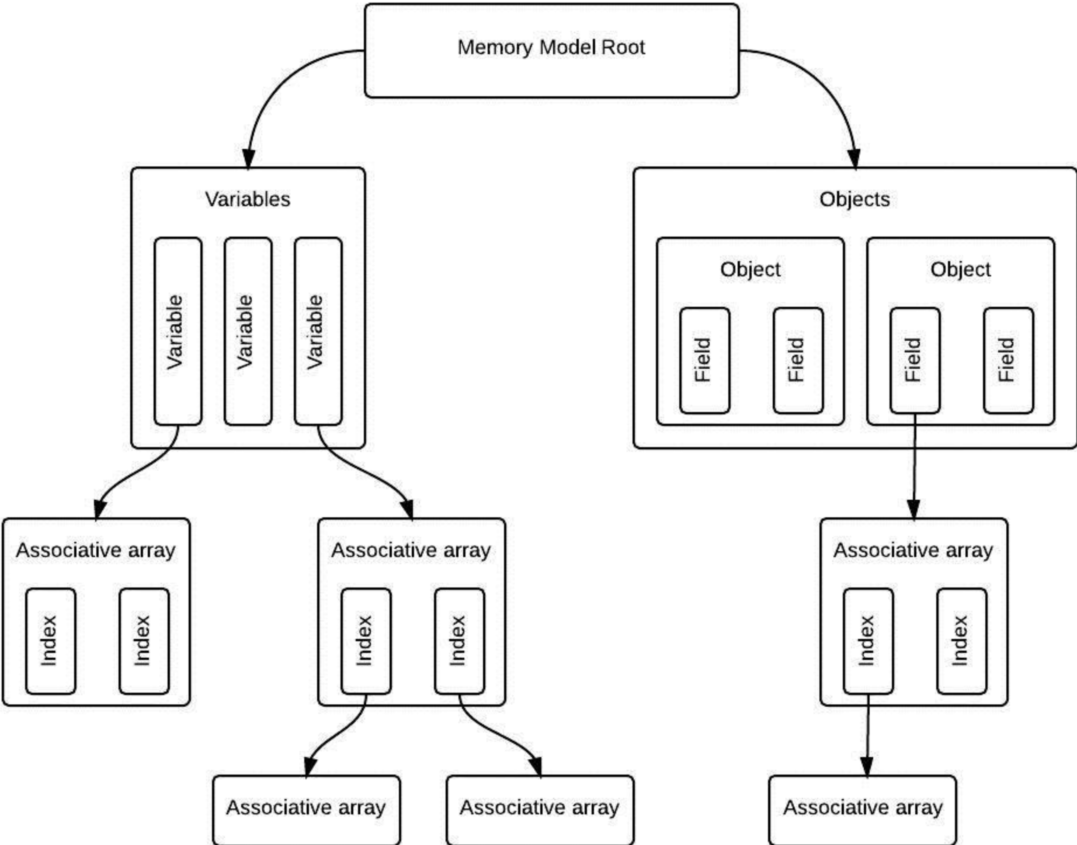
3.8.2.4 Concept of memory tree

The first theoretical concept of copy memory model was that every access path points to a different location. Memory model then have to distribute updates to all aliases - every location contains all data which may appear there.

Even that this behavior ensure write-read semantics it causes several problems with objects. Programmer can easily make cyclic dependence between references or create access paths with huge number of chained descendants. Existence of sequence of referenced objects slows down computation when whole tree has to be copied to another location. Of course programmer can do this using arrays and aliases but this is not typical for their usage. In contrast to quite common structures like linked lists, trees or graphs of object instances.

Because of this issue the copy memory model uses only access paths with arrays. Object fields introduces new root of access path (the first root is list of variables). Memory model now has to process PHP path with indexes and fields into sequence of access paths when there is some field contained.

Using index sequence as pointers to memory locations causes that every memory location cannot contain more than one array. So if there is a possibility that in some location may appear more than one array, memory model has to merge these arrays together. This is not necessary for objects. Because of reference semantics one memory location can contain more than one object.



- Hierarchy of memory tree in copy memory model -

3.8.2.5 Implementation of memory representation

Memory representation needs to support these constructs:

- Variables
- Control variables
- Arrays
- Objects
- Temporary variables
- Uncertain (any) locations for variables, indexes and fields
- Aliasing
- Call stack

Classes *MemoryIndex*, *MemoryStructure*, MemoryData, *ArrayDescriptor* and *ObjectDescriptor* has been introduced in order to implement these functionalities.

Memory model itself is represented by *Snapshot* class.

Memory index

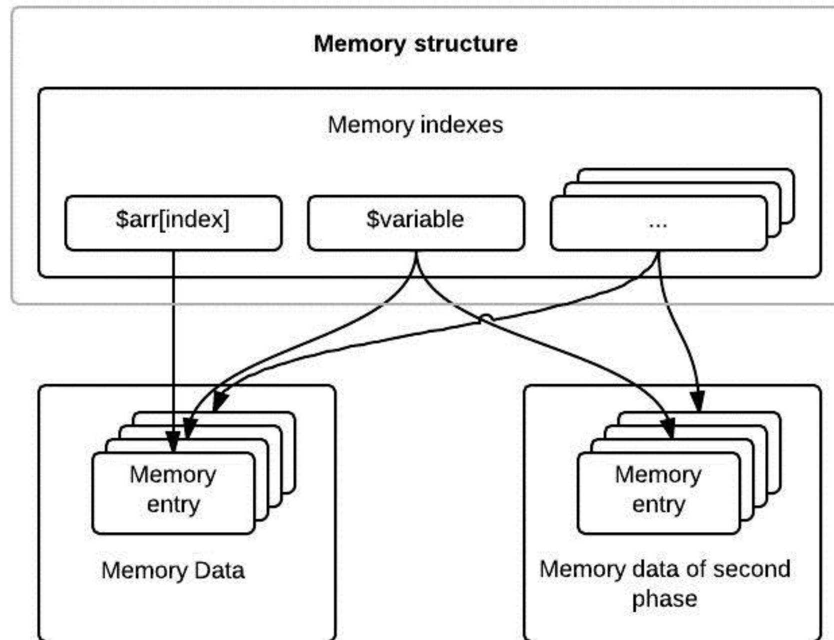
Core element of memory representation is *MemoryIndex* class. This class implements access path described above. These objects are used across whole copy memory model anywhere where it is necessary to link between some memory locations within snapshot or even between two different memory model instances (merge algorithm).

Indexes also allows to separate description of structure and current memory data. This is the main reason of existence of *MemoryIndexes*. Because of these indirect pointers it is not necessary to change whole structure of memory tree if some memory location is changed.

Consider a situation when link between two locations would have been implemented as direct reference between them. The simplest change might have started cascade of changes across whole memory model. Indirect indexes brings difficult navigation but change of one location does not have any effect to its memory index so the pointing object can be still used.

Indexes also allow analysis to use different sets of data for a single structure in next part of analysis. This division also brings opportunities for future research in order to optimize access to structure or data.

Memory model main class combines structure and data implementations together - classes *MemoryStructure* and *MemoryData*. This classes allow to browse or update memory tree using theirs public interface. Memory model automatically creates new memory locations when analysis wants to write some data into them.



- Using memory indexes to divide structure and data -

Variables

Every access path in PHP is rooted in some variable. The variable is the simplest access path which is commonly used across whole PHP program. On the side of memory model there has to be a mechanism to map each variable to appropriate memory location and allow analysis to access each variable by its name. Memory model also needs to separate handling variables, fields, indexes and other special memory locations.

All of these is possible by introducing indirect indexes based on access paths. On the side of memory entry there is just simple associative container which maps raw variable name to memory index. Anytime when analysis needs to manipulate some variable, there is simple lookup to this map with constant complexity. Given index can be processed by the same routines no matter the type of memory location.

The same approach can be reused for other parts of access paths. No matter if the accessed memory location is variable, control variable, index or field. The memory model has to map raw name into memory index using associative hash map. This approach demands additional memory for hash map and storing collection of memory indexes but speeds up analysis because is not necessary to create new index object when is needed. Memory indexes are also implemented in such a way that the comparing of two indexes is much faster when both indexes are point to the same object instance.

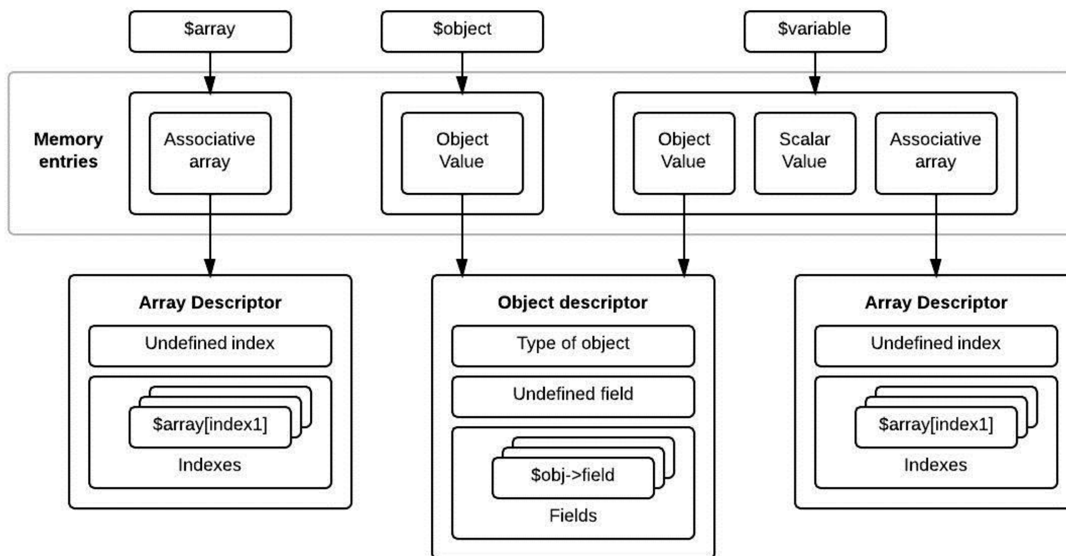
Array and object descriptors

In contradiction to previous paragraphs the division of memory model to structure and data cannot be strict. It is because structure of memory tree is defined by chain of associative arrays as mentioned above. Usage of objects also brings modification of structure because every object field points to some memory location which is part of structure.

Analysis framework uses special values to support associative arrays and objects. These values are empty objects which can be found in memory entries of memory location and are used as pointers to objects and arrays descriptions. Copy memory model uses special descriptor objects which contains metadata for each object and array (type of objects, associated location and list of used indexes and fields). Memory structure instance contains associative container which maps pointing objects to appropriate descriptor. Descriptor can be used to access some memory location, traverse the memory tree or when some algorithm needs to manipulate with array or object.

Using pointing object also brings the same advantage which was mentioned in the paragraph about memory indexes. Indirect pointers prevent update cascade when some array or object changes its structure - it is not necessary to walk through all memory entries and insert new value of array into it.

As was mentioned above the way the structure is assembled requires to have only one array in memory location. Otherwise ambiguous memory locations can occur. When analysis needs to insert two arrays into the same memory location it is necessary to merge these arrays into one. In case of objects there is no limitation. Reference semantics of objects allows to have multiple objects in the same memory location.



- Using descriptors for object and arrays -

Control and temporary variables

Collection of control variables allows analysis to store special values within the memory model. Control variables are very similar to the standard variables so the memory model uses the same routines to work with them as with the normal variables.

Temporary variables are separated list of memory locations for inner usage in memory models - to store data which are not assigned with any memory location and to copy data between two memory locations.

The first case handles cases when programmer creates new array without the variable - just as a parameter of function call. This array is not rooted in any variable but copy memory model needs to handle any memory location the same way.

Second case prevents to interfere between read and assigned memory locations. This can happen when PHP code wants to assign into memory location path which is prefix of the memory location path which is read. This may happen because of several reasons - cyclic assign or wrong usage of aliases. To prevent this memory model just creates deep copy of data to temporary location and then runs assign algorithm from this new location.

Uncertain memory locations

The analysis is not always capable to determine target memory location for data (any value as index in array). When this happens uncertain memory location is used.

There is two levels of uncertain reads and writes. In the first level analysis figure out that there is only limited number of writings targets - for example two valid indexes where data can be written to. This is the simplest way - memory model just needs to provide weak update of two different memory locations (in contradiction to strong update of a single location). There is no need for special support in snapshot structure. This is the example of code where this case happens:

```
if (?) { $x = 1; }
else { $x = 2; }
$arr[$x] = "value";
```

The second possibility is that analysis is not capable to determine the index at all. In this case memory model has to provide weak update of all existing memory locations in the given level of an array, object or even root variables (when double dollar is used). This is fully controlled by update algorithm and there is no need for special support. Except of updating previously created locations the update command may write into location with any name. Memory model needs to support even this functionality because when analysis needs to read from new location it is possible that there can be some data. Consider this code:

```
$arr[$_POST['id']] = "value";
```

The result of consequent read of any index of the array `$arr` should include the value `"value"`. To support this behavior there are special locations on each node of memory tree (array, object, variable collection). This *any* locations are places where the

data for any memory locations are stored. Algorithms of memory model then uses these locations to get data from undefined locations.

Aliasing

Aliasing technique is similar to references in C++. Aliases are widely used when passing arrays as function arguments. Because of copy semantics it is much faster to pass huge array as alias then copy it every time. This is also useful when programmer needs to use some variable as output parameters.

These are legit usages of alias mechanism. The same is quite common even in the world of C++ and other languages with copy semantics and pointers. Of course PHP with its pointer operand allows to create alias between any pair of variables or even fields or indexes. There is no limit and unaware usage of this mechanism can cause cyclic dependence or various side effects if alias is created only in specific branch of control flow graph.

Memory model needs to allow this functionality. Alias can be established between any pair of memory locations. The existence of alias can be uncertain or even the target of aliasing can be ambiguous or even unknown. *Must* or *may* aliases was introduced to model this behavior. Memory model needs to ensure proper updates of all aliased locations.

From the structural point of view it is necessary to store informations about aliased locations. Implementation of this behavior is straightforward. Every memory location has two lists of memory indexes for must or may aliased memory locations. Algorithms of memory model use this information to provide update of all aliased locations.

Call stack

Call stack is memory structure which supports global and local variable contexts for function calls. The main problem is that variables in different contexts do not directly interfere. However two memory locations can be connected indirectly via aliases so two variables can be changed across call stack (e.g., local variable can be alias of parameter of the function). There is a possibility to access global variables from any function. The levels of memory stack has to be separated because of all this. But memory model also has to allow changes across multiple levels of memory stack.

To implement this behavior, the structure of memory model contains implementation of call stack for all memory locations rooted in some variable. Root of each memory index contains information about type of access path (variable, control, temporary) and level of call stack. Two indexes with same name and type can differ by the call stack. Object fields and nested paths are excluded from call stack because of reference character of objects.

Algorithms of memory model use memory stack. Memory model creates new level when there is a new call in snapshot and clears it when call is over. Algorithms typically work on local level but routines for handling aliases automatically updates memory locations across the stack when is necessary.

Alias mechanism is also used for handling global variables. When programmer in PHP imports global variable into local namespace memory model just creates alias between local and global variable and uses alias algorithm to handle this connection.

3.8.2.6 Operations over the memory model

The second part of memory model is to allow analysis to read and to update memory structure which was described above. Interface of any memory model is provided by several implementations of abstract classes defined in the analysis framework. In the case of copy memory model there are *Snapshot, SnapshotEntry* and *DataEntry* classes which implement *SnapshotBase* and *ReadWriteSnapshotEntryBase* abstract classes.

Snapshot instance represents full state of memory in a single program point. It contains full memory structure, data and implements set of methods to support memory transactions. Snapshot entries are special objects which are instantiated by snapshot object. Each of these instances represents ticket to some memory access path. Through the usage of this ticket analysis is able to read or modify set of memory locations which satisfies given access path.

Access paths provided by snapshot entries are different than access path used to identify memory locations. Every memory location has unique access path composed by indexes where each index can have single value or special any value. Snapshot entries models PHP access paths with root variable and sequence of mixed indexes and fields. Every segment can contain uncertain names - zero for any value, one for certain indexes or multiple values when analysis can modify multiple locations. Single snapshot entry can identify multiple access paths in the snapshot.

Memory collectors

Mapping snapshot entries to memory locations is not trivial. Because of reference semantics of objects, aliases, uncertain or unknown locations can happen that to provide operation it is necessary to strongly or weakly update many different memory locations.

Every operation is split into two different parts. Firstly it is necessary to collect all memory locations which can be accessed by the algorithm. Memory model has to traverse the memory tree using the specified access path. Secondly read or update of these memory locations is performed.

When the operation just reads memory data collector traverse the memory tree using *breadth-first search* algorithm and collects set of indexes where the data can be read from. When memory model does not contain memory location for some part of access path unknown memory location is used to continue the traversing. When algorithm cannot proceed even using the undefined location the branch is cut and undefined value is inserted into the result set. Read collector also do not need to traverse aliases because all valid data are copied between all memory locations.

Updating collectors has to be different. They are also traverse the tree by BFS algorithm. In contrast to read collector the update collector has to create new memory location when analysis wants to update location which is not created. When collector creates new memory location it cannot just create blank location. Because of modifying unknown memory location there can be some value which may be written into the new

location. Collector copies all values from undefined memory location when some new location is created on the same level of memory tree.

It is not just creating new variables, indexes of arrays or fields of objects but when there is no array or object in the traversed location the element has to be created at first. Special semantics of PHP allows to create implicit objects when program writes value into some field to undefined variable and same for indexes of associative array. When the variable contains scalar value the new object or array is not created and warning is raised. Memory model needs to model this behavior and creates new entities just for variables which may be undefined. So even when variable contains object where the traverse can continue the memory model has to create new object when there is possibility of undefined memory location.

Update collector also has to traverse memory tree not only by direct descendants of processed locations but also using must or may aliases.

After collecting of memory entries and preparing source and target memory location, the other part of operation can proceed. The other part is reading or updating algorithm itself which gets the collected locations and do requested work.

Read operation

Reading is straightforward. Algorithm just goes through all collected memory locations and gets all data associated with all of these locations. An extra undefined value is added into output set when the collector considers that there may be some undefined location where is read from.

Update operation

Update operation is much more interesting. The character of *may* and *must* locations determines that some of the locations has to be updated strongly and others weekly. Strong update means that given value has to be written into the memory location. Memory model has to delete content of memory location if there is any - all arrays has to be removed and structure has to be cleared. Then algorithm has to copy the given value into memory location. This value can contain associative array so the algorithm needs to traverse its structure and copy array and its data into new location.

Weak update means that in some program branches there can be modification of this location but when the program will proceed by different branch the value won't be written. The old value has to stay in the target location and new values has to be merged with them. And also both source and target can contain associative array. As result there has to be only one array per location so the structure and data of arrays has to be weakly merged.

Assign alias operation

Assigning aliases is similar to update operation. In this operation memory model needs to strongly or weakly copy data from aliased locations to targets. This operation uses two update collectors to collect locations of sources and targets. Both can be in two variants - may or must.

Operation itself provides update which was described above. Then it connects given locations by alias connection so on update one of aliased variable is also updated the other one. Weak or strong semantics determines whether to create must or may alias.

Merge operation

Merge operation is invoked when analysis needs to merge two snapshots of memory model when two or more branches of program point comes together. Because there can be any change of data or structure in any branch, merged snapshot can contain different data. In order to allow analysis to continue from the meeting point it is necessary to combine all snapshots to a single one.

Merge operation finds all memory locations in all merged models and combines its data together. Because there can be some location which is not defined in some snapshot, merge algorithm combines known and unknown locations or adds undefined value when there is not appropriate data in some snapshot.

As result of merge operation there is a new object with merged structural data - variables, arrays, objects and definitions of classes and functions. This new object and object with data can be used in target snapshot.

3.8.2.7 Future work and optimization of memory model

This implementation of copy memory model is not optimal. During the implementation of Weverca tool there were some theoretical challenges to determine complexity of analysis itself. This implementation of memory model is a first working implementation and it is a base for future work. There are several challenges to reduce the complexity of used algorithms - parallelism, laziness, sharing data containers and more. These concepts can be used to optimize copy memory model to reduce memory and time complexity.

Also the write-read semantics can be subject of future research. As was mentioned above during this example implementation there were some architectural decisions between copy and reference semantics. It would be interesting to determine whether the strong write-read semantics is possible for real web applications. And if not there is always a possibility to modify algorithms to get more precise results.

3.9 Function resolver

Function resolver provides functionality to resolve direct and indirect function, method and static method calls. It produces warning when trying to call an inaccessible methods or function or method which doesn't exist.

When function or method is being called, control-flow graph and program point graph is created on demand and then added into the flow.

Analyzer uses sharing program point graphs based on information stored in memory model. Shared program point graphs will be used when:

- method is called at least 3 time in one recursion
- if recursion depth is more than 10, sharing program point graphs will be used in case that function is called for second time

Function resolver takes care of initialization of variables on beginning of function call:

- initializes function arguments

- increases call depth in local control variable *.callDepth*
- set current script full file name into local control variable *.currentScript*
- set current function value into local control variable *.currentFunction*
- set called object type into local control variable *.calledObject*
- stores number of calls of current function into variable *.calledFunctions*
- fetches super global variables from global container

Function resolver also handles control variables when calling eval or includes.

Return values from calls, includes and evals are copied from local context. Before handing the value to framework, function hints are applied on return variable. In this moment analyzer has an opportunity to decrease variable values handling eval and include depth.

3.9.1 Native analyzers

Native analyzers are singleton classes which provides information about native constants (*NativeConstantAnalyzer*), native objects (*NativeObjectAnalyzer*) and native functions (*NativeFunctionAnalyzer*). During construction of these instances, all native information are read from xml files.

Information provided by native analyzers:

- global constants and their values
- native classes, fields, static fields, initialization values and base classes
- function and method argument types and return type based on PHP documentation

For every function and method native analyzer provides a delegate which based on type information models current function or method. Native analyzers also holds information about functions which are sanitizing or reporting. Reporting function or method is method which reports a warning if some input values are tainted.

3.9.1.1 Type modeling of native functions

Modeling delegate check number of arguments and their types and report warnings. Then it takes all tainted flags and copies them to return value and to arguments passed by reference.

3.9.1.2 Type modeling of native methods

It is very similar to native functions but it also sets called object fields to any typed values. It copies flags from arguments and object fields to object fields, arguments passed by reference and return value.

3.9.1.3 Particular implementation of native functions

Tool also provides particular implementation for native function to make analysis more precise. These function are provided by class *NativeFunctionsConcreteImplementations*. In case none of the arguments are abstract, particular implementation is used.

3.9.2 Function hints

Function hints are created when returning value of function. They are stored in Dictionary inside Function resolver. Function hints clean flags from return value based on php documentation comments. Comment line to be considered a hint has to match this regular expression: `^[ \t]**?[ \t]*@wev-hint[ \t]+sanitize[ \t]+(HTMLDirty|SQLDirty|FilePathDirty|all)`

3.10 Expression resolver

3.10.1 Overview

Expression resolver is part of program that fixpoint algorithm needs for its work. It evaluates all constructs of language that behave like expression: variables, constants, literals, operations, assignments etc. It is not a compact component, but rather a service that knows to evaluate lots of different elementary expressions. The resolver simulates behavior of PHP runtime, but it is extended since it must be able to evaluate not completely accurate data. *ExpressionEvaluator* is the main class of the resolver.

Expressions in analysis can give arbitrary number of possible values, because static analysis cannot always determine one precise value in every program point. Result of evaluation is stored in the *MemoryEntry* object representing one place in memory. There can be particular values of one of the PHP types (*Boolean, integer, float, string, object, array, resource* or *null*). However, very often, it is not suitable representation of data.

There are introduced abstract values of each basic type that represents any possible value of the given type, and one universal typeless abstract value. For more accuracy, there are number intervals too. Usually, values sent to resolver method have general base type *Value*. To inspect the right particular type, lot of evaluation is implemented in visitor patterns derived from *PartialExpressionEvaluator* class that has method for every value representation in analysis.

3.10.2 Conversions

Conversion of values of one type to another type is a basic operation of each language. It is applied explicitly by casting or implicitly in other expressions. PHP has dynamic and weak type system and it permits conversion between every pair of types. However, not all conversions are properly defined. For instance, conversion of object into integer makes no sense. Such conversions are implementation-defined and then cannot give exact value, but return abstract value of the proper type.

There are also conversions whose result depends on a value of the expression, for instance result of conversion from floating-point number into integer is abstract if value does not fit to integer. Conversion of abstract value behaves as the conversion of all values that the abstraction represents and its result contains a superset of all these values. Conversion of a particular value into Boolean is always defined because of conditions, anyway, the analysis has no problem with abstract Boolean.

3.10.3 Unary and n-ary operations

Operation in analysis differs from the PHP runtime in such a way that operands are not one particular value, but set of possible values. The idea of evaluation of unary operations is extremely simple. In *UnaryOperationEvaluator* visitor class, the particular method is called for every possible value depending on its run-time type and then it is transformed depending on the type of the operation into resulting set. It can process arithmetic, bitwise and logic operation and casting, that is kind of unary operation. *IncrementDecrementEvaluator* class can resolve increment and decrement separately, because we must distinguish between their prefix and postfix form. There is only one n-ary operation - concatenation. Phalanger recognizes it as n-ary operation, but fortunately, it can be simulated by sequence of binary concatenations.

3.10.4 Binary operations

Binary operations are complicated both in source code size (occupy most of the resolver implementation) and time and memory requirements for analysis. There must be a code resolving every combination of an operation and two types of operands. The simplest mechanism that would resolve both operand at once is triple-dispatch, so we cannot use basic visitor pattern that is way to implement the double-dispatch only. The resolver solves it by using of two levels of visitor pattern, each for one operand.

There is *BinaryOperationEvaluator* visitor pattern class that initially determines type of the left operand. However, it does not perform any computation, but chooses another visitor derived from *LeftOperandVisitor* class and send it the value of detected type. There must be implementation of this class for every value type. The particular class knows the type of left operand and it suffices to determine the type of the right operand, which is the same mechanism as unary operations.

Since it is necessary to perform a binary operation for each pair of possible values of both operands, the number of possible values in result may grow quadratically. The straightforward solution is to limit the number of values. The resolver performs simple reduction of values, where every type has limit of maximum values and if it exceeds, all values are widened into an abstract value. In any case, there is an area for tuning, especially for specialized programs that works with narrow range of values, strings for instance.

Binary operation can be divided into four groups of similar operations: Comparison, arithmetic, logical and bitwise operations. Operations of particular operands are either defined by PHP or undefined and then an abstract value must be return. Operation with abstract values usually does not give good result. Intervals are an exception, they may give very precise result in comparison with general abstract number types.

Logical operations and comparison have good characteristics from the perspective of the analysis. They reduce both operands with arbitrarily number of values to just one Boolean value (where an abstract Boolean is allowed, of course). Moreover, logical operation is evaluated conditionally depending on value of the first operand.

Comparison has a bit confusing in PHP, it compare types by many different manners. Many abstract values are difficult to compare.

Arithmetic operations are meaningfully defined only for numbers, other values are converted. Operations with intervals are very nice, if operations succeeds, it creates coherent interval. Arithmetic is inaccurate only if result overflows or underflows, because then the integer is converted into floating-point number. Bitwise operations are the hardest to predict. They take only integer operands and others are converted. If one operand is abstract, it is very difficult to calculate something reasonable, because the resulting numbers may not constitute any inherent interval.

3.10.5 Variable resolving

Resolver does not manage variables, this is the job of a memory model, but it can access current context of analysis and reads from and writes to variables. If variable is read/written, resolver just does some additional operation: It checks visibility of class members, whether index is applied to array or string value and eventually reports warning. It creates new array if variable that contains `NULL` value is accessed by index (i.e. code "`$var = NULL; $var['index'] = 'value';`" results in creating of array similar to `$var = array('index' => 'value');` construction). PHP language allows access to variable by expression (it is called Variable variables). Resolver converts values of expression to names of variables and types.

3.10.6 Creating new values

The model may get new values from outside (static variable - `$_GET`, `$_POST`, return values of function calls) or by initialization. Scalar types are initialized to literals. Objects are created by construct `new` with a type. There are initialized all non-static properties of an object, including properties defined in all ancestors of the appropriate type. If type is expression, its values are converted to string and used as names of types to create multiple objects.

An array value can be created by `array()` language construct. It takes list of key/value pair of parameters, the key value is arbitrary. And just the keys may cause problems during initialization. During the initialization, PHP keeps default index that is zero at the beginning. The elements are stored in sequence, one after another. If key is not given, the default one is used and then it is incremented. If key is given, but it is not represented by number (or another type representing the number, e.g. number in string), it is used as index, but default one is not incremented. However, if key is integer, it replaces the default one. Since a key is expression, it can represents any number of possible values. Moreover, the key may be merged with default one. As a result, the initialization of array may be very inaccurate and the more parameters there are, the less accurate it is.

3.10.7 Type declarations

During type declaration analyzer:

- converts classes to common format with native classes (*ClassDecl*)
- stores static variables and constants into memory model
- copies information from base classes
- checks inheritance of methods and throws warnings when error occurs
- checks implemented interface methods
- checks for multiple field, method and constant declarations
- checks interface constants and copies them into declared class

3.10.8 Static variable storage

Static variables are in associative array stored in special global control variable called *.staticVariables*. Every index contains another associative array where static variables are stored. Every class stores values of its own static variables. Variables from parent class are stored only in parent class and variable from child class points on parent class variable with alias.

3.10.9 Global constant storage

Global constants are also stored in global control variable *.constants*. PHP constants can be defined as case sensitive or insensitive. Case insensitive constant can be declared using method `define`. Case sensitive constants are stored with prefix “`#`” and case insensitive constants are stores with prefix “`.`”. For used defined constants inserting are constants retrieving is responsible class *UserDefinedConstantHandler*. Native constants are read from *NativeConstantAnalyzer*.

3.10.10 Class constant storage

Native class constants are stored in *NativeObjectAnalyzer*. User defined class constants are stored directly in global control container. Every constant is stored in variable *.class([class lower case name])->constant([constant name])*. For example constant *a* in class *x* is stored in variable *.class(x)->constant(a)*.

3.10.11 Foreach

`foreach` construct is similar to statement `for`, more precisely, it is a particular type of `for` loop, that traverses elements of an array. The advantage for analysis is such that we know the number of cycles. And not only that, we also know a value of iteration variable in every cycle. So we can use a simple approximation such that we merge all values from given array (or arrays) into iteration variable and enter into body of the loop only once.

3.10.12 Special constructs and build-in functions

There are some expression-like constructs that cannot be classified well. Some of them, such as `eval`, is processed by framework itself. Constructs like `exit`, `empty`,

`isset`, `echo`, or `instanceof` is evaluated by expression resolver. `echo` does nothing in analysis, but makes sense for taint analysis.

3.11 Flow resolver

3.11.1 Overview

Flow resolver is designed to give more precise assessment of the variables used in the branching commands like `IF` and `SWITCH`. This is not necessary because we already have certain idea about the possible values of all the variables used in the branching command, but as the branching commands are used very much in a typical code, it is still useful to have as precise evaluation of the variables as possible. It makes the calculation of the fixpoint easier. The Flow resolver should not add any more values to the domain of the variable. It should restrict existing assessment according to the condition used. For example it we have a condition like

```
IF ($a > 3) {…}
ELSE {…}
```

and before this block we know, that the variable `$a` may contain any integer value, in the positive branches of the `IF` command we might restrict the domain of the variable just to the interval (minInt; 3).

We can even say something about the negative branch of the `IF`. The domain of the variable would be <3; maxInt).

The goal of the Flow resolver is also to tell whether the condition can be satisfied. Based on that, the calculation of the domain of the variables is triggered. It makes no sense to calculate the domain of the variables used in the condition if we know that the condition cannot be satisfied.

3.11.2 How does it work

Flow resolver is implemented in the namespace *Weverca.AnalysisFramework.FlowResolver*. The main class is called *FlowResolver* and it is derived from the *FlowResolverBase*. In this class there is method *ConfirmAssumption*, which is used to start the calculation over the given condition. The condition is given in the instance of class *AssumpitionCondition*, which encapsulates everything we have to know about the condition. That is not only the condition statement itself, but also the form of the condition. Based on the situation we might want the parts of the condition to be all true, some true, exactly one part true, exactly one part false, some false, or all to be false.

The parameters of the method also includes an instance of the class *EvaluationLog*, which is used during the calculation to get known values of the parts of the condition and its variables and an instance of *FlowOutputSet*, which is used as a memory context of the condition and its branches. *FlowOutputSet* is also used as an output parameter, where the assessments for each calculated domain of any variable will be stored.

The *FlowResolver* splits the condition into parts and work with each part separately and then merges the results together using either union or intersection according to the form of the condition. Some conditions are already made from parts when the method is called. These parts are in property *AssumptionCondition.Parts*. This is used for `SWITCH`. For example when we have a `SWITCH` construction in the code and we are calling *FlowResolver* for the default branch, we would have for each case of the switch one condition part and the form of the condition would be "None". The merging operator would be union, because we need all of the parts of the condition not to hold.

Some conditions are not split to the parts when the Flow resolver is called. These are conditions used in IF-like constructs. For example `IF ($a > 3 && $a < 5)`. In this case Flow resolver breaks the condition into the parts itself and recursively evaluates each part. The operator used for merging is chosen according to the logical operator used in the condition. For `AND` the operator is intersection and for `OR` the operator is union.
The intersection merging operator is not an intersection in mathematical sense. In example above it is, but if we consider condition like `IF ($a > 3 && $b < 5)`, we have nothing to intersect, because we cannot intersect domains of two different variables.

Because of these different approaches to the merge operation, Flow resolver uses its own memory context implementer in class *MemoryContext*, which calculates the intersection and union of given domains of variables. After the calculation is finished, the calculated results held in the *MemoryContext* will be assigned to the *FlowOutputSet* given to the *FlowResolver*.*ConfirmAssuption* as a parameter.

3.11.3 Exceptions

Flow resolver stores information about visited try blocks in global control variable *.catchBlocks*. Each time framework visits try block, all information about catch blocks associated with current try block are pushed to stack stored in *.catchBlocks* variable. If end of this block is visited, data from stack are removed.

After exception is thrown, program finds program points, where analysis should continue. In catch block the stack is unrolled to proper state and the catch variable is assigned.

3.11.4 Includes

Analyzer reads all possible included files, creates program point graph and adds new branches into the flow. For every included file analyzer uses variable *.includedFiles* to store information about number of includes of current file. At the end of every included file analyzer decreases number of include calls in memory model. If one include is called at least 3 times in one "recursion" we use shared program point graph.

3.11.5 Eval

Eval is resolved very similarly to include with some small differences. Eval doesn't use shared program point graphs. Eval call stores eval call depth in control variable *.evalDepth*. If depth of eval recursion is greater than 3 analyzer will not resolve any more

evals and produces warning. This limitation was made to avoid infinite program point creation in source codes like this:

```
$a='eval("$a")';
eval($a);
```

3.11.6 Future works

Flow resolver supports only operators `=`, `!=`, `<`, `<`, `<=` and `>=`. Generally, when dealing with non-numeral values of variables, there is only little to say about the domains of the variables for the Flow resolver in the current version. It would be possible to implement a “reverse evaluation” of some methods used in the conditions.

For example consider condition like `IF (abs($a) == 2)`. It is clear that the domain of variable `$a` must be {-2, 2} if the condition holds. But for this to work, the Flow resolver would have to know how the method `abs` works. Constructs like this are now not supported. There are many built-in methods for which the evaluation proposed here would be useful.

3.12 Adding support for new PHP features

Depending on the type of feature, the required modification may include:

- syntax elements, which change the flow of program (e.g. new type of cycle) - support has to be added to project *Weverca.ControlFlowGraph* (class *CFGVisitor*), method Visit[new element name].
- syntax elements, which doesn’t change the flow of program - for every element a new program point needs to be created (subclass of *ProgramPointBase*). Method *flowThrough* needs to be implemented and from this method some type of resolver method is called, which handles analysis of current feature.
- other features (e.g. new magic functions), framework and control-flow graph do not have to be changed, support has to be added into a function or another resolver.

3.13 Web

Weverca web is a simple user-friendly GUI for Weverca tool, which can be used as an alternative to Weverca console application. It is written in ASP.NET MVC 4 under .NET framework 4.5. Framework version is required because Weverca is written in mentioned version. It is not needed for the web interface itself.

3.13.1 Project settings

The timeout for the analysis is configurable in the file web.config located in the root directory of the web application. The setting is called *AnalysisTimeout* and its

integer value tells how much time in milliseconds the analysis can take. It is not recommendable to alter anything else in the configuration file.

3.13.2 Debugging of the project

Weverca project is written using Microsoft Visual Studio 2013. It can be easily opened in Visual Studio 2012 or newer. Visual studio 2010 or older cannot be used because of the version of the .NET framework.

The project can be debugged in the same way as a desktop application using IIS express, IIS, or Development Server of Visual Studio 2012.

3.13.3 Deployment

Weverca web can be hosted in IIS 8 (Microsoft Internet Information Service) or newer, which is part of Windows Server 2012 or never.

To deploy the project first publish it using Visual Studio then create a web application in IIS using the path where published web is located and .NET 4.5 application pool in integrated mode. There is no need to alter configuration file, apart from setting up non-default analysis timeout, which is 10 000 ms.

Weverca Web uses NLog for logging any error, which might occur during the analysis. Logging is configured in the file Nlog.config. There is a path for the logs in this file. The application pool must have rights for writing to the target folder; otherwise the logging will not work. The application will work correctly even if the permission is set incorrectly, but no logs will be produced. See http://nlog-project.org/ for more details about NLog.

3.13.4 Future works

So far it is not possible to upload multiple files for analysis in Weverca Web GUI, which would be useful.

Also it would be nice to link the code editor on the result page with the results. For example when moving mouse over the code corresponding results might be displayed near the cursor.

4 Conclusion

The tool created in this project supports relatively modern version of PHP (5.1) and it is able to process constructs specific to PHP and other dynamic languages, such as JavaScript. Weverca also computes metrics judging the quality of given source code.

Implemented control-flow graph was adjusted to specific constructs of PHP. Project includes analysis framework which allows the user to implement their own analysis and modify implementation of existing analyses in a simple and flexible way. Tool implements two different and independent memory models, so user can choose memory model for analysis. Another possibility is to replace these memory models with some other implementation. This provides option to run analysis with different memory models and study its behaviors. Framework also provides an opportunity to implement new second phase analyses for gathering additional information about given source code.

Weverca tool also gives user information about parse errors and possible runtime errors. Usually this information is available to programmer during code interpretation, but after Weverca integration into some IDE, this information can be available sooner. As a result programmer can save significant amount of time while developing and testing PHP applications.

The main goal, to create a software which can show programmer possible security weaknesses, was reached.

Possible future work:

- Replace Phalanger 3.0 with newer version of Phalanger, which supports newer versions of PHP
- Analysis accuracy can be improved:
    - Widening precision can be improved
    - Flow revolver's variables assumptions can be expanded
- Native function and object analyzer can use more accurate modeling of native functions and native methods than type modeling, which is used for most functions
- Modeling of non-native functions
- Assume code annotations for enhancing the scalability of the analysis
- Implement more analyses in second phase (e.g detection of dead code, path-sensitive validation of errors found by analysis)
- Integration into development environments

5 References

[1] N. Jovanovic, C. Kruegel & E. Kirda (2006): Pixy: a static analysis tool for detecting Web application vulnerabilities. In: S&P'06, IEEE. New York, NY, USA, pp. 336–346.
[2] Fang Yu, Muath Alkhalaf & Tevfik Bultan (2010): Stranger: An automata-based string analysis tool for PHP. TACAS'10.
[3] Etienne Kneuss, Philippe Suter & Viktor Kuncak (2010): Phantm: PHP Analyzer for Type Mismatch. In: FSE'10.
[4] Phalanger project - The PHP Language Compiler for the .NET Framework, http://phalanger.codeplex.com