Static Analysis for the PHP Language

Etienne Kneuss
etienne.kneuss@epfl.ch

January 12, 2010

Abstract

This report presents the work that was done to implement a static analysis tool for the PHP programming language. The kind of analysis done by the compiler or by the multiple development environments are very limited. This tool aims at providing further feedback to a developer by checking for multiple bug conditions or mis-uses of the language and should reduce the risk of encountering fatal errors at runtime after deployment of PHP based application.

1 Introduction

The PHP programming language is a compiled language. However, the compilation is done by default each time a certain PHP file is requested. This means that the compilation has to be fast and hence can't do much analysis and checks. Most of the checks are thus made at runtime. The purpose of this tool is to allow the developer to check its work for runtime-errors, without actually running its application.

2 Design

This tool consists of a custom PHP compiler, translating the code into structures such as Abstract Syntax Trees (AST) and then Control Flow Graphs (CFG). Fixpoint analysis is then performed on the control flow graphs. This compiler is divided into five distinct parts:

1. Lexing
2. Parsing and Abstract Syntax Trees
3. Semantic Analysis
4. Control Flow Graphs
5. Type flow Analysis

2.1 Lexing

The lexer uses a modified version of JFlex[1]. It is thus written in java. The original PHP compiler uses flex as scanner. The modification done to JFlex is the implementation of flex's `yymore()`, which currently lacks in JFlex. This function is used to tell the scanner that the following rules should be appended to `yytext` instead of overwriting it. The tokens generated are closely related to those PHP 5.3.0 would generate. One notable difference is that we translated every single-char tokens to their respective name.

2.2 Parsing and Abstract Syntax Trees

The parser, as mentioned before, is based on a modified version of CUP. CUP is a java based parser generator which uses a syntax similar to yacc, the parser generator used in the original PHP compiler. The version of CUP used for this project differs from the original one in two points:

1. The original CUP cannot handle large syntax files. Indeed, since CUP directly writes the parsing tables as properties of the generated class, the size of the class properties can become too large for *javac*. The PHP grammar being quite big, the resulting parser class couldn't be compiled. To solve the problem, CUP was modified to provide an option specifying that the multiple tables should be written into files and loaded at runtime.

2. To avoid the troubles of referencing scala classes, the parser generator was modified to translate the application of the production into a simple tree which could then be translated by a Scala class entirely.

2.3 Semantic Analysis

Some checks can be done directly by looking at the AST. Indeed, PHP provides a wide range of features, some of them being considered nowadays as bad practice. This tool emits notices for the following issues:

- Non top-level declaration: PHP allows a developer to conditionally declare functions or classes. This not only generates problems with the following analysis, but also induces some performance hits on servers equipped with so-called opcode cachers. Those opcode cachers are responsible for caching the intermediate–or compiled version of each file, function and class. The goal being to speed up the process by reducing the number of compilations required per request. This cannot be done easily if those declarations are conditional.

- Call-time pass-by-ref: a function accepting a reference[2] has to be defined as such, but PHP allows the developer to pass a reference at the time of the call, even to a function not declared to receive one. This feature is deprecated, and can cause unwanted side-effects.

[1] JFlex: http://jflex.de
[2] PHP References: https://php.net/references

- Non-trivial include calls: the `include`[3] statement allows a developer to execute the given file in the current scope. The argument representing the file to include can be dynamic. This tool will try to resolve dynamic expressions that are most commonly used to be able to extend the analysis to that file. In case the expression is too dynamic, that include call will be ignored. Any combination of those expressions are considered as trivial dynamic expressions for include calls:
  - the concatenation operator: "."
  - the `dirname()`[4] function, used to retrieve the parent directory of the path passed as argument
  - constants
  - class constants
  - string

  With such trivial expressions, the analyzer will look for it in the multiple include-paths entries and directly attach the corresponding AST to the current one.

- Dynamic object properties: PHP allows dynamic references to an object property using a variable or expression (e.g. `$name = "a"; $obj->$name` instead of `$obj->a`). This is usually considered as bad practice since arrays are usually preferred for such tasks.

- Dynamic variables: PHP allows to reference a variable using either a variable, or an expression[5]: (`$$var` or `${'prefix'.$name.foo()}`).

- Assignations in conditional expressions: assignations in PHP return the value assigned, they are hence valid expressions inside conditional expressions. However, history tells us that, most of the time, this is an actual typographic error replacing the assignation operator `=` with the comparison operator `==`. This tool will thus emit a warning if such expression is found inside an `if()` or `for()` condition. We exclude `while()` on purpose as there is a common use-case where assignations are done directly inside the `while()` expression[6].

Another part of the semantic analysis consists of validating the usage of identifiers such as variables, functions or classes. The goal is to ensure that no obvious semantic errors such as inheritance cycles, or visibility inconsistencies exist.

2.3.1 API Importation

By default, PHP comes with a very dense library of functions and classes. In fact, the main extensions that are shipped with PHP consist of more than 2'500 functions and classes. Being able to correctly represent this internal API is a key factor to obtain useful analysis results. This API is stored in an external

[3]PHP Include: http://php.net/include
[4]PHP dirname: http://php.net/dirname
[5]PHP Variable variables: http://php.net/variables.variable
[6]PHP mysql_fetch_assoc: http://php.net/mysql_fetch_assoc

XML file, allowing easy modifications and also do not require a re-compilation. Additionally, a `--apis` command line option is available to specify a list of API files that can be imported into the symbol tables. This is especially useful for large projects as it allows focused analysis.

2.3.2 Annotations

One of the language aspect that is limiting the analysis is that PHP does not support type specifications–or type hinting for everything. In fact, it is only accepted in function arguments, and the type specification is limited to *arrays* and *objects*. As a result, the type of the return value cannot be specified, and arguments that take scalar values such as *booleans* or *integers* cannot be specified as such.

However, many project use formatted comments that they can then render into an API documentation using some tools, such as php documentor[7]:

Listing 1 Block comments as annotations

```
<?php
class A {
  /**
   * @var int
   */
  public $plop;

  /**
   * @param string $str
   * @return int
   */
  public function length($str) { return strlen($str); }
}

?>
```

The analyser will look in the AST for comments directly preceding functions, methods or properties declarations. It will then try to extract such type informations and inject it as type hints in the AST. The analyzer will then simply use them like normal type hints during the type checking phase.

2.4 Control Flow Graphs

In fact, the AST is only the intermediate step that only let us do trivial checks. To analyse more complex aspects of a program, this tool will use control flow graphs (CFG) which are derived from AST's. Control flow graphs provide a model of the flow of values through the different control structures. Because PHP is dynamically typed, and since variables are not declared, it is impossible to infer and check types solely based on the structural representation of the code. For such tasks, we require CFG's to be able to reproduce the flow of types that would happen in an execution of the code. The main focus of this

[7] phpDocumentor: http://www.phpdoc.org

analysis will be on analysing type flows. In the CFG, each vertex correspond to a program state between statements, and each directed edge represents the application of a statement.

3 Analysis

3.1 PHP Overview

Before trying to see how to analyse PHP code, it is important to understand how PHP works, this is a small overview of the features PHP provides that are relevant to this analysis.

3.1.1 Types

PHP is a weakly, dynamically typed language, meaning that types are not specially associated to variables, but to values. A variable may hence hold values of different types in its lifetime. It is also weakly typed as it allows automatic–or implicit type conversions when performing an operation on values of different types. This is specified as part of PHP as "type juggling"[8]. In PHP, we have the following types:

- **Booleans**
- **Integers**
- **Floating point numbers**
- **Strings**
- **Arrays**: Arrays in PHP are ordered hashmaps, allowing either strings or integers as keys. They can contain values of any type that may be mixed.
- **Objects**
- **Resources**: Resources represent special data types such as file handle, database connection links. They can be of different types. PHP extensions that define resource types are responsible of handling them appropriately.
- **Null**: This is the default type to any undefined or uninitialized variable.

3.1.2 Object Oriented Programing

PHP supports OOP as of PHP4, but many features have later been added into PHP5. The OO model as well as the syntax used is closely related to a subset of what Java offers: public/protected/private visibility, single inheritance, with the support of interfaces. PHP provides object properties, object methods, static properties, static methods and class constants. You can dynamically define an object property in PHP, its visibility will default to public. However, you cannot dynamically define methods, static members, or constants.

[8]PHP Type Juggling: http://php.net/language.types.type-juggling

3.1.3 Limitations on Analysis

Many of the dynamic features available in PHP will get in the way of a sound analysis. Here is a non-exhaustive list of such features:

- **autoload**[9]: PHP allows you to trigger a function call in case an undefined class was used. This function call is usually used to subsequently load the appropriate class at runtime. This feature allows programmers to only load classes on demand.
- **__call**[10]: If a method call is defined in an object, calling an undefined method of that object will instead call the `__call` method, with the name of the original method, and the arguments used.
- **__callStatic**: This is similar to `__call` but works for undefined static method calls.
- **__get**[11]: This method will get called in case an access to an undefined object property is done. The value returned by the method will correspond to the property value.
- **ArrayAccess**[12]: The ArrayAccess interface allows an object to be used as an array. For example, if we have `class Foo implements ArrayAccess`, then `$foo = new Foo; echo $foo['index']` is accepted.
- **Dynamic accesses** PHP allows multiple ways to dynamically access variables, classes, members... Here are a couple of examples:
  - `$$n`: Accesses the variable named by the value of `$n`.
  - `$$$n`: Accesses the variable named by the value of `$$n`.
  - `new $n()`: Constructs an object of the class named by the value of `$n`.
  - `$c::$n()`: Call the static function named by the value of `$n` on the class named by the value of `$n`.
  - `$n()`: Call the function named by the value of `$n`.
- **eval**[13]: The `eval()` construct allows a developer to evaluate the code passed in as a string.

3.2 Typeflow Analysis

As said earlier, the main focus of this analysis tool is on types. Even though PHP is dynamically typed, and provides very few indications of types. It is still interesting to try to infer types and check for type safety.

[9] PHP Autoload: http://php.net/autoload
[10] PHP Overloading: http://php.net/__call
[11] PHP Overloading: http://php.net/__set
[12] ArrayAccess Interface: http://php.net/arrayaccess
[13] PHP Eval: http://php.net/eval

3.2.1 Type model

We now provide a detailed description of every elements of our type model:

- *TAny* ($\top$): Top of our lattice, represents any type
- *TNone* ($\bot$): Bottom of our lattice, represents no type
- *TBoolean*: Any boolean type.
- *TFalse*, *TTrue* : True/False types, used for conditional type filtering (See 3.4.1).
- *TInt*: Integers
- *TFloat*: Floating point numbers
- *TResource*: Resources. We make no distinction between resources of different types. This could be improved in the future.
- *TNull*: Null value.
- *TString*: Strings.
- *TUnion*: Union of multiple distinct types, represented as the infix $\cup$ operator. The result of this union of types depends on the operands, it will be described into further details in its own section.
- Arrays: Arrays are represented using a quite precise model, it is structured as follows:
  - A set of ($index \rightarrow type$) associations for every well-defined array entries. The rationale behind having such a precise representation is that PHP arrays are often used as containers for multiple named values. Unlike other languages like C, PHP allows values in arrays to be of multiple types. Those values are then used independently. It is thus important to keep track of the types associated to each index.
  - An optional array-wise default type. This type represents the possible contamination of an array via a dynamic access. This type is called the "polluted type". For example, the code `$a[$foo] = 2;` will have the following effect on a well-defined array: $\forall (e \rightarrow t) \in entries : t = t \sqcup TInt$. Also, we have $polluted\ type = polluted\ type \sqcup TInt$. We can also represent this default type as ($? \rightarrow type$). A side effect of having a non-empty polluted type will be to turn off any notices related to undefined entries. In fact, any lookup for an undefined entry in an array will default to the polluted type, if any, or issue a notice for a potentially undefined index.

  We note that, even though PHP are *ordered* hashmaps, we totally ignore to model the order as it is not something that people commonly rely upon. Additionally, we argue that if somebody is traversing an array in order, potentially all types of the array will get visited. Hence, iterating over an array will yield values of type equal to the union of all possible types of the array.

  Two types are used internally to represent arrays:

- *TArray*: The general type representing the model described above.
- *TAnyArray*: This represents an array being the supertype of all arrays. It is in fact an alias to an array whose only entry is $(? \rightarrow TAny)$.

- Objects: Objects are represented differently than other types. The reason is that when passing, assigning or returning objects PHP will pass the object reference. For example, in `$a = new Foo; $b = $a; $a->foo = 2`, both *$a* and *$b* represent the same object, hence both variables are affected by the object modification. We need to reproduce that level of indirection as part of our type model for objects. This is done by introducing object references as types, mapping the real object in an object store. That way the type can be copied around and propagated to different values and still point to the same real object type. We thus have two different levels of types:
  - **Object references:** This is the type to which any object value is assigned.
  - **Real objects:** The real object is pointed to by the object reference. This is the actual object type storing mappings to methods, properties and constants. It is stored in the object store and is not directly exposed as types of values.

  Real objects are decomposed into two object types:
  - *TRealClassObject*: Representing an object of a defined class. This type will have, like arrays, a index-type mapping for properties. Since object properties can be dynamically defined, we also have a so-called polluted type for properties.
  - *TRealObject*: This is very similar to *TRealClassObject*, the only difference is that this object carries no class information. It is mostly used for type constructions. For instance, in the code `$a = new A; $a->b = 2;` the statement `$a->b = 2;` will merge the type of `$a` which is $TRealClassObject(A)(\cdots)$ with a constructed $TRealObject(a \rightarrow TInt)$. This will result in $TRealClassObject(A)(\cdots, a \rightarrow TInt)$.

3.3 Type Lattice

A lattice is usually defined by three things: a partial order relation ($\sqsubseteq$), a join operation ($\sqcup$), and a meet operation ($\sqcap$). As noted earlier, the $\sqsubseteq$ relation is the subtyping relation, we have that $A \sqsubseteq B \Leftrightarrow A <: B$. The join operation $A \sqcup B$ returns the smallest type C such that $A \sqsubseteq C \wedge B \sqsubseteq C$. As for the meet operation we can ignore it in our case since it is not necessary for top-down analysis.

3.3.1 Definition of $\sqsubseteq$

Those are, in order, the rules that apply to determine whether $t_1 \sqsubseteq t_2$:

1. $\forall t_2 \in types : TNone \sqsubseteq t_2$
2. $\forall t_1 \in types : t_1 \sqsubseteq TAny$

3. $TFalse \sqsubseteq TBoolean$
4. $TTrue \sqsubseteq TBoolean$
5. $\forall(t_1 : TObjectRef) : t_1 \sqsubseteq TAnyObject$
6. $\forall(t_1, t_2 : TObjectRef) : t_1 \sqsubseteq t_2$ if all the following rules apply ($o_1 = store(t_1.ref), o_2 = store(t_2.ref)$):
   (a) $o_1.class <: o_2.class$ if o_1 and o_2 are real class objects
   (b) $o_1.pollutedType \sqsubseteq o_2.pollutedType$ if any
   (c) $\forall f \in o_3.fields : f \in o_1.fields \wedge o_1.fields(f) \sqsubseteq o_2.fields(f)$
7. $\forall(a : TArray) : a \sqsubseteq TAnyArray$
8. $\forall(a_1, a_2 : TArray) : a_1 \sqsubseteq a_2$ if all the following rules apply:
   (a) $a_1.pollutedType \sqsubseteq a_2.pollutedType$ if any
   (b) $\forall e \in a_2.entries : e \in o_1.entries \wedge a_1.entries(e) \sqsubseteq a_2.entries(e)$
9. $\forall(u_1, u_2 : TUnion) : u_1 \sqsubseteq u_2 \Leftrightarrow \forall t_2 \in u_2 : \exists t_1 \in u_1 : t_1 \sqsubseteq t_2$
10. $\forall t_1 \in types, (u_2 : TUnion) : t_1 \sqsubseteq u_2 \Leftrightarrow \forall t_2 \in u_2 : t_1 \sqsubseteq t_2$
11. $\forall(u_1 : TUnion), t_2 \in types : u_1 \sqsubseteq t_2 \Leftrightarrow \forall t_1 \in u_1 : t_1 \sqsubseteq t_2$
12. $\forall t_1, t_2 \in types : t_1 \not\sqsubseteq t_2$ If none of the rules above applies

3.3.2 Definition of Join ($\sqcup$)

Those are, in order, the rules that apply to determine the result of $t_1 \sqcup t_2$:

1. $\forall t \in types : t \sqcup TNone = TNone \sqcup t = t$
2. $\forall t \in types : t \sqcup t = t$
3. $\forall(t : TObjectRef) : t \sqcup TAnyObject = TAnyObject \sqcup t = TAnyObject$
4. $\forall(t_1, t_2 : TObjectRef) : t_1 \sqcup t_2 = TUnion(t_1 t_2)$. As we know that at this point, the types must be different, which means they point to different objects.
5. $\forall(t : TArray) : t \sqcup TAnyArray = TAnyArray \sqcup t = TAnyArray$
6. $\forall(t_1, t_2 : TArray) : t_1 \sqcup t_2 =$ an array where the polluted type is the join of both polluted types, and the entries is the union of both entry sets. Note that for an entry present in both arrays, the join of its types is taken.
7. $\forall t_1, t_2 \in types : t_1 \sqcup t_2 = TUnion(t_1, t_2)$. If none of the above applies, we get a union of types.

3.4 Analysis Mechanics

As we know, this analysis is based on point-wise lattices. A lattice is a structure providing a partial order between its elements. This is specially useful to represent types (the order relation being subtyping) which is what we use in our case.

The role of a point-wise lattice is to keep track of a lattice value, or type in our case, for every values. We will denote those point-wise lattice elements as type environments. A type environment is basically a mapping between values and types.

The analysis algorithm starts by assigning a base type environment to every CFG vertexes, or a global bottom ($\bot_g$) value, indicating whether such a vertex has been visited yet. Then, the algorithm starts with the entry of the CFG, and will visit the graph in order. For each edge $e = v_1 \rightarrow^s v_2$ where s is the statement, we can calculate the environment at $v2$ coming from e_i by applying a transfer function on v_1 with s. This will represent the type environment in which v_2 will be in case that edge e was taken to get to v_2. Of course, one vertex may have multiple entry edges and so, to calculate the actual type environment of v_2 we will have to join every environment env_i defined as $env_i = transferFunction(v_i, s_i)$ where $(v_i \rightarrow^{s_i} v_2) \in edges$. This type environment join operator ($\cup_e$) is defined as follows:

- $\bot_g \cup_e E_i = E_i$ and $E_i \cup_e \bot_g = E_i$
- For $E_i \cup_e E_j$ where $E_i \neq \bot_g$ and $E_j \neq \bot_g$, we construct $E_u = E_i \cup_e E_j$ using the following set of rules: First, we define $E_i(v_j)$ as the type associated to the value v_j in the environment E_i. We have that $\forall v_k \in values(E_i) \cup values(E_j)$:

$$E_u(v_k) = \left\{ \begin{array}{ll} E_i(v_k) \sqcup E_j(v_k) & v_k \in values(E_i) \wedge v_k \in values(E_j) \\ E_i(v_k) \sqcup TNull & v_k \in values(E_i) \wedge v_k \notin values(E_j) \\ TNull \sqcup E_j(v_k) & v_k \notin values(E_i) \wedge v_k \in values(E_j) \end{array} \right\}$$

  Note that the join operator $\sqcup$ used with two types represents the *join* operation on our type lattice, as described earlier in the type lattice section.

We see that traversing the CFG only once will not be sufficient. For instance, in an loop, in the first vertex in the loop, the entry edge coming from the end of the loop is not known when we enter the loop the first time. Multiple passes are hence required! In order for this algorithm to terminate, we check whether we actually changed the environment. This so called "fix-point" algorithm will stop whenever no change has been made to any environment during a complete traversal of the graph. By design, the type lattice join operation will not generate types that will always change, as it can only go "up" in our lattice. However, we need to take special care in the type transfer function that it doesn't generate differences that would jeopardize the termination. For instance `$a = new MyClass` cannot be made to create a new object reference for every traversal! To solve this case, we set the reference value to the program point.

3.4.1 Conditional Type Filtering

So far, the analysis features are quite common and could be applied to most languages. This is however insufficient to give a good analysis. The reason is that most internally defined PHP functions will return `false` or `null` in case of error. For a function normally returning an integer, this means that any return value of this function should be typed $TInt \cup TBoolean$. However, this would lead to massive amount of false positives. Consider the following code example which can be seen in most procedural scripts dealing with mysql queries:

Listing 2 Fetching query results

```
<?php
$result = mysql_query(..);
if ($result) {
    while($row = mysql_fetch_assoc($result)) {
      ..
    }
}
?>
```

`mysql_query` either return a result resource or false in case the query failed. `mysql_fetch_assoc` takes a resource, and return an array representing the row, unless it reached the end of the result set in which case it will return false. Naively checking types would give us many false positives:

- `mysql_fetch_assoc`'s argument would be of type $TBoolean \cup TResource$, which would fail with a type mismatch saying that it expects a $TResource$.
- Inside the while body, `$row` would still be of type $TBoolean \cup TArray$, generating a false positive for any usage of `$row` as an array.

The solution to this problem is in two parts. First of all, it was necessary to improve the precision of the boolean type and introduce $TTrue$ and $TFalse$. That was, a function can be described as returning $TResource \cup TFalse$. From those types, it becomes quite trivial to filter incompatible values when taking a branch.

For instance, with `if ($result)` the CFG will create two branches, one for $\$result = true$ and one for $\$result \neq true$. In the first branch, we can filter out types of `$result` that represent strictly false values, which includes $TFalse$ and $TNull$. The same applies for the second branch, in which we filter out strictly true values, namely $TTrue$ and $TResource$. In our previous example, the type of `$result` inside the if body would thus be filtered to $TResource$ which is correct. Moreover, it will also filter assigns as conditionals, meaning that in `while($row = mysql_fetch_assoc($result))`, the while body will have $TResource$ for type of `$row`. In summary, this simple filtering trick allows for precise prototyping with reduced false-positives!

4 Limitations and Future Work

Most serious PHP applications out there are often based on some heavy frameworks that may require a lot of "magic" features to work. Analysing such projects would currently yield a flood of false-positive, totally hiding any potential real problem.

Lots of features are not completely checked or integrated in the analysis, such as exceptions, namespaces, or closures. In general, there are still lot of areas of the language for which the analysis is not done or hasn't been properly tested.

This tool still fails at analysing one "core" feature of PHP, which is *references*. As of PHP5, the usage of references is in most case obsolete or even harmful. However, it is still used by lots of projects, for reasons such backward compatibility or performance improvements.

5 Conclusion

Doing precise analysis on PHP is hard, and that is because the language has never been developed with such concerns in mind. We can however see the potential applications of such tool, like direct integration as part of an IDE or plugged to a source repository as a quality assurance tool. The fact that only few developments are aimed at providing analysis for PHP can possibly be explained by its difficulty. As a consequence, the PHP world still has a lot of room for analysis tools such as the one described here, which makes its development quite interesting and motivating.

References

[1] A.W. Appel *Modern Compiler Implementation in Java* Cambridge University Press (2002)

[2] Benjamin C. Pierce *Types and Programming Languages* The MIT Press (2002)

[3] N. Jovanovic, C. Kruegel, E. Kirda *Precise Alias Analysis for Static Detection of Web Application Vulnerabilities* Technical University of Vienna

[4] S. Hangal, M.S. Lam *Automatic Dimension Inference and Checking for Object-Oriented Programs* Standford University

[5] http://lara.epfl.ch/dokuwiki/compilation: example_efficient_code_for_conditionals