Static analysis of dynamic scripting languages

Draft: Monday 17th August, 2009 at 10:29

Paul Biggar
Trinity College Dublin
pbiggar@cs.tcd.ie

David Gregg
Trinity College Dublin
david.gregg@cs.tcd.ie

Abstract

Scripting languages, such as PHP, are among the most widely used and fastest growing programming languages, particularly for web applications. Static analysis is an important tool for detecting security flaws, finding bugs, and improving compilation of programs. However, static analysis of scripting languages is difficult due to features found in languages such as PHP. These features include run-time code generation, dynamic weak typing, dynamic aliasing, implicit object and array creation, and overloading of simple operators. We find that as a result, simple analysis techniques such as SSA and def-use chains are not straight-forward to use, and that a single unconstrained variable can ruin our analysis. In this paper we describe a static analyser for PHP, and show how classical static analysis techniques can be extended to analyse PHP. In particular our analysis combines alias analysis, type-inference and constant-propagation for PHP, computing results that are essential for other analyses and optimizations. We find that this combination of techniques allows the generation of meaningful and useful results from our static analysis.

1. Motivation

In recent years the importance of dynamic scripting languages — such as PHP, Python, Ruby and Javascript — has grown as they are used for an increasing amount of software development. Scripting languages provide high-level language features, a fast compile-modify-test environment for rapid prototyping, strong integration with database and web development systems, and extensive standard libraries. PHP powers many of the most popular web applications such as Facebook, Wikipedia and Yahoo. In general, there is a trend towards writing an increasing amount of an application in a scripting language rather than in a traditional programming language, not least to avoid the complexity of crossing between languages.

As scripting languages are used for more ambitious projects, software tools to support these languages become increasingly important. Static analysis is an important technique that is widely used in software tools for program understanding, detecting errors and security flaws, code refactoring, and compilation. However, PHP presents unique challenges to this analysis. In addition to highly dynamic features, it has very complicated semantics, whose edge cases combine in complex ways. It is dynamically typed, and programs provide no type- or alias-information to the analysis writer. Worse, even simple statements can have hidden semantics which must be modelled in order to allow conservative program analysis.

PHP's complicated features include:

1. run-time source inclusion,
2. run-time code evaluation,
3. dynamic, weak, latent typing,
4. duck-typed objects,
5. implicit object and array creation,
6. run-time aliasing,
7. run-time symbol-table access,
8. overloading of simple operations.

Some of these features have been handled by earlier work. (1) and (2) have been addressed by string analysis [29] and profiling [11], early work [18] has been performed on alias analysis (6), and a number of other problems (3, 8) have also been touched upon [30]. However, large sections of the PHP language are left unmodelled by this previous work, including important cases relating to arrays, and essential features such as object orientation.

In particular, it is notable that all previous analyses have been non-conservative. Previous analyses have been aimed at bug-finding, but their solution have not been suitable for purposes such as compilation or optimization. This may be due to the difficulty of providing a conservative analysis for PHP, which we believe we are the first to do.

PHP's feature set allows simple expressions to have hidden effects based on a value's run-time type. These hidden effects are exacerbated by PHP's run-time aliasing, and their detection is hampered by PHP's dynamic typing. Their presence makes it very difficult to determine a simple list of definitions and uses of variables within a function.

Let us consider an assumption made by earlier work [16, Section VI.B], which does not perform type-inference or model PHP's weak typing and implicit array creation:[1]

> if the expression `$a[2]` appears somewhere in the program, then `$a` certainly is an array.

Unfortunately, this was untrue for their analysis on PHP4, and is less true in PHP5. In a reading context, `$a[2]` may read from a string, an array, an uninitialized variable or any other scalar value (in which case it would evaluate to `NULL`). In the writing context, it might write into a string, assign an array value or convert a NULL value or uninitialized variable into an array. If `$a` was an ***int***, ***real***, ***resource*** or ***bool***, then the assignment would generate a run-time warning, and the value would be unaffected. In PHP5, if `$a` was an

[1] In PHP syntax, `$a[2]` indexes the value in the variable $a.

object, then the `get` object handler would be called, if present, or else there would be a run-time error.

It should be clear that many of PHP's features make analysis difficult, and that traditional analyses are not adequate to analyse these features. In addition, existing work on PHP omit many features of PHP. In this paper we present novel and elegant abstractions and extensions to traditional compiler analyses to enable these analyses. Our analysis is whole-program, optionally flow- and context-sensitive, and combines alias analysis, type-inference and constant-propagation. It is an ambitious analysis which models almost[2] the entire PHP language. We believe we are the first to handle such a large portion of the PHP language, as we are able to analyse a large number of features which were not handled by previous analyses.

In Section 2, we present the semantics of PHP from the perspective of program analysis, particularly discussing features which make analysis difficult. In Section 3 we discuss previous analyses for PHP, and how their limitations prevent analyses of the whole PHP language. We highlight areas where the unconventional semantics of PHP have led to edge cases not modelled by previous work, and how traditional analysis for C or Java is insufficient to model PHP. In Section 4 we describe our novel analysis, how it solves both the problems highlighted in Section 2, and the deficiencies in all previous work to date. Our experimental evaluation and discussion is presented in Section 5.

2. PHP language behaviour

PHP has no formal semantics, no rigorous test suite, and an incomplete manual. As such, there is no definition of its semantics, except for its reference implementation.[3] We have previously described a technique [3] for creating a compiler for such a language.

Although PHP is very different from languages like C, C++ and Java, there are a great deal of similarities to other scripting languages such as such as Javascript, Lua, Perl, Python and Ruby.

PHP offers many language features which makes program analysis difficult. In order to prime the description of our analysis in Section 4, we present first an informal description of the behaviour of PHP 5. We have omitted behaviour which has no effect on program analysis.

2.1 PHP Overview

PHP has evolved since its creation in 1995. It was originally created as a domain specific language for templating HTML pages,[4] which was implemented in Perl. It is influenced by Perl, and has similar syntax and semantics, including a latent, weak, dynamic type system, powerful support for hashtables, and garbage collection. PHP programs are typically web applications, and PHP is tightly integrated with the web environment, including extensive support for database, HTTP and string operations.

PHP 3 introduced simple class-based object orientation, which used hashtables in its implementation. Since tables are copied during assignment, syntax was introduced to allow variables to reference other variables, so as to pass arrays and objects to functions and methods.

PHP 5 introduced a new object model, with new assignment semantics for objects. This allowed object pointers to be copied into new values, creating a second means of passing objects by reference. However, the old reference semantics remain, and are still required for passing arrays or scalars by reference.

[2] See Section 4.9 for the limitations of our approach.

[3] PHP's reference implementation can be downloaded from `http://www.php.net`.

[4] PHP was originally named "*Personal Home Page/Forms Interpreter*".

2.2 Dynamic typing

Typically, in statically-typed imperative languages such as C, C++ and Java, names are given to memory locations. Variables, fields and parameters all have names, which typically correspond to space on the stack or the heap. Each of these names is annotated with type information, and this is statically checked.

PHP is instead dynamically typed, meaning type information is part of a run-time value. The same variable or field may refer to multiple run-time values over the course of a program's execution, and the types of these values may differ.

2.3 Arrays

Arrays in PHP are maps from integers or strings to values. The same array can be indexed by both string and integer indices. Arrays are not objects, and cannot have methods called on them. They are implemented as hashtables in the reference implementation, and we refer to them as ***tables*** from here-on-in.

Tables form the basis of a number of structures in PHP, including objects and symbol-tables. Symbol-tables are a very interesting case. PHP variables have subtle differences to languages like C, where they represent stack slots. Instead PHP variables are fields in global or local symbol-tables, which form a map of strings to values.

PHP provides run-time access to symbol-tables. The `$GLOBALS` variable provides access to any global variable by name. The local symbol-table can be referenced using ***variable-variables***. A variable-variable is means of reading or writing a variable whose name might only be known at run-time. Listing 1 demonstrates reading and writing the variable `$x`. We note that the presence of a run-time symbol-table becomes apparent due to variable-variables: it is possible to read and write to variables which would violate PHP's syntax check.

```
$x = "old value";
$name = "x";
$$name = "new value";
print $x;        // "new value"
```

Listing 1: Example of the use of variable-variables.

PHP's built-in functions occasionally have access to the symbol-table of their callers. `compact` and `extract` respectively read and write from the caller's symbol-table, bundling and unbundling values from variables into arrays. The variables that are read and written-to can be chosen based on dynamic values. A number of other functions also write the the `$php_errormsg` variable in case of error.

The `$GLOBALS` variable, which points to the global symbol-table,[5] can be passed to array functions, iterated over, introspected or cleared. In fact, the user can even unset the `$GLOBALS` variable possibly, possibly preventing direct access from the user.

Many of the behaviours of arrays are shared by other features which are implemented using arrays. For example, reading from an uninitialized field in a PHP array will return `NULL`, and so will reading an uninitialized value from a symbol-table as a result. These behaviours are also shared by PHP's objects.

2.4 Objects

PHP's class system is a mishmash of other type systems. Unlike a number of other scripting languages, there is a static class hierarchy. Once a class is declared its parent class cannot be changed, a fact which simplifies analysis. In addition, classes may not change their methods after they are declared. Similarly, once an object is

[5] which holds the `$GLOBALS` variable...

__call
__get
__invoke
__toString
__set
__isset

(a) ***Magic methods***

call_method
cast_object
get
get_constructor
get_method
has_dimension
has_property
read_dimension
read_property
set
write_dimension
write_property

(b) C object handlers

Figure 1: Selection of object handlers.

instantiated it cannot change its class, and as a result its methods may not be changed either.

Otherwise, PHP uses ***duck-typing*** [21]. Fields have no declared type, and may be added and deleted from an object at any time. As such a class does not denote a particular memory layout. Like symbol-tables, the reference implementation implements objects using tables, and the semantics reflect that. For example, an uninitialized field (that is, accessing a field of an object which does not yet have that field) evaluates to `NULL`, just like arrays and symbol-tables.

A simple class declaration and example is shown in Listing 2. Note that the field `num` is not declared anywhere.

```
class MyInt {
    function value () {
        $this->num = rand ();
        return $this->num;
    }
}

$int = new MyInt(5);
$newint = $int->value ();
```

Listing 2: A simple class demonstrating field use without declaration.

Unlike arrays and symbol-table, objects have an extra layer of magic, which was introduced in PHP5. An object's class can define methods to handle some operations. Figure 1a contains a selection of allowed magic methods. `__get`, `__set` and `__isset` are called if inaccessible fields are accessed, `__call` is called if inaccessible methods are accessed and `__invoke` is called upon an attempt to invoke the object.

The `__toString` method is of particular interest. Strings are fundamental to PHP, and there are a large number of places in which the `__toString` method may be called, should an object be passed instead of a string. Examples are the concatenation operator ('.'), interpolated variables in strings, printing, or using any of PHP's built-in functions which expect strings. We note that a `__toString` method may be called from deep within an otherwise opaque library, if the library expected strings passed to its interface.

A final layer of magic lurks behind the scenes. Many PHP libraries are implemented using PHP's C API [3]. Objects passed to or from library functions may have special C object handlers. These can be called in some circumstances in which magic methods cannot — such as when the object is read or written to — or accessed using array syntax. Figure 1b contains most of these handlers. Typically, if defined, they replace the standard behaviour of some PHP construct, for example accessing an object using the array syntax (`*_dimension`) or accessing fields (`*_property`).

Since the libraries written using the C API may have access to a large portion of the internals of the reference implementation, C object handlers can in theory change anything.

2.5 References

A reference between two variables establishes a run-time alias, where variables share the same value. This has strong similarities to references in C++. However, PHP's references are mutable — they can be created, used, and destroyed again at run-time. Unlike C pointers, a PHP reference is bidirectional, as there is only one value being referred to by multiple names. These multiple names may be passed widely throughout the program, and the same values may be referenced by many names, including function parameters, or field of objects or arrays, or symbol-tables entries.

```
1   function a() {
2       $a2 =& $GLOBALS['x1'];
3       if (...)
4           $a1 =& $a2;
5
6       b(&$a1, $a2);
7   }
8
9   function b($fp1, &$fp2) { ... }
10
11  a();
```

Listing 3: Example of dynamic aliasing in PHP. References creation can be at run-time, and in some cases occurs conditionally. This is a combination of Figures 7 and 8 from [18].

Listing 3 demonstrates PHP's dynamic aliasing. On line 2, `$a2` becomes a reference of `$GLOBALS['x1']`, that is, the global variable `$x1`. Line 4 shows the creation of a possible reference.

To complicate matters further, although a function may be declared to take a parameter by reference, a caller may optionally also pass its actual parameter by reference. Thus, either the callee or the caller can specify that a parameter is passed by reference. Line 6 shows a call to the function `b`, where the first parameter is passed by reference in the caller, and the second parameter is passed by reference due to the signature on line 9.

2.6 Scalar values

PHP has the following scalar types: ***int***, ***real***, ***string***, ***bool***, ***resource*** and ***null***. It is not possible to add user-defined scalar types.

int and ***real*** are wrappers for the C types ***long*** and ***double***. ***bool*** is a simple boolean type which may be `true` or `false`. ***string*** is a scalar type in PHP, as there is no character type for it to aggregate. However, array syntax may be used to modify portions of a string, complicating analysis of arrays.

Much of PHP's standard library is written in C. A ***resource*** provides a way to wrap C pointers in user-code, after they are returned from standard libraries. PHP user code cannot create resources, and operations on them are largely not meaningful. Resources do not otherwise affect our analysis, and so shall not be addressed further.

PHP's `NULL` value closely resembles the unit value [22] in Scheme. Scheme's unit type has exactly one value, also called unit. Similarly, PHP's NULL type has exactly one value, also called `NULL`. However, comparisons between PHP's `NULL` and those of other languages are muddied by PHP's weak-typing.

PHP allows definitions of constant values, similar to C's `#define` construct. However, ***constants*** are dynamically defined in PHP, by calls to the `define` function. As such, they are defined at run-time, not compile-time. Although a constant may not change its value once it is defined, a constant may be conditionally defined, may be defined late in the program, or may be defined from a user value. Constants may only be defined using scalar values.

2.7 Weak typing

A major feature of PHP's type system is that as well as being dynamically typed, it is also weakly typed. That is, conversions between types often happen automatically and behind the scenes.

This is due to PHP's heritage, since it was originally designed as a language for creating web applications. At the time, it was useful to take strings directly from the user, as shown in Listing 4 and using them directly as other types. Though this practice is frowned upon because of potential string injection vulnerabilities, weak typing is still an integral part of the language.

```
if ($_GET["age"] > 25)
    echo "...";
```

Listing 4: Example of weak typing. `$_GET` fetches a string value from the user, whose integer value is compared with 25.

2.8 Type-coercion

PHP values can also be cast from one type to another explicitly by the programmer. The PHP manual [25] goes into great detail about the behaviour of casts. As such, we will only focus on those that pertain to program analysis.

Casting a scalar to an array creates a new array containing that scalar (using the key "*scalar*"). A cast to an object is similar, except that the result is an object which is an instance of "*stdClass*". The exception to these rules is the `NULL` value, which is cast to an empty structure. It is not possible to cast between object types (or otherwise change the concrete type of an object), or from a scalar to a particular object type.

Casting an object to a string will invoke the object's `__toString` method, as discussed in Section 2.4.

2.9 Operators

There are two equality operators, == and ===. The former uses weak rules, considering for example the integer `5` to be equal to the string `"5"`. The === operator is stricter, requiring the type of the values to also be equal.

PHP has detailed rules on the weak equality of values, described in the PHP manual. Many of these rules were collected by experimentation on the existing behaviour in some version of PHP, but they appear to be unchanged since their behaviour was first added to the manual. An interesting property of the == operator is that it is not transitive, as can be seen in Listing 5. Other comparison operators, including arithmetic and bitwise operators, have similar quirks.

```
print "zest" == 0;          // true
print 0 == "eggs";          // true
print "zest" == "eggs";     // false
```

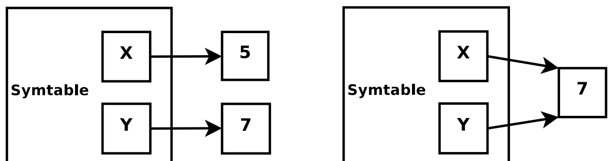
Listing 5: Example of intransitive equality operations.

2.10 Assignment

There are two forms of assignment in PHP. The most straightforward, denoted `$a = $b`, is an assignment by copy, which copies the value into new memory. In Listing 6, `$x` is assigned the value 5, after which `$x`'s value is copied into `$y`. Consider the run-time memory representation after the copy, shown in Figure 2a. As explained in Section 2.3, `$x` and `$y` are symbol-table entries, each of which point to run-time values, each set to 5.

```
                    int *y = malloc(sizeof(int));
                    int *x = malloc(sizeof(int));
$x = 5;             *x = 5;
$y = $x;            *y = *x;
$y = 7;             *y = 7;
```

Listing 6: Assignment by copy, with comparable C code.



(a) Assignment by copy (see Listing 6). (b) Assignment by reference (see Listing 7).

Figure 2: PHP assignment memory representation.

Assignment ***by reference*** creates a reference between the two variables, causing them to share a single value. In Listing 7, `$x`'s value is shared by `$y` on line 3, and both `$x` and `$y` are defined on line 4. Figure 2b shows the memory representation after line 4.

```
1                   int *x = malloc(sizeof(int));
2  $x = 5;          *x = 5;
3  $y =& $x;        int *y = x;
4  $y = 7;          *y = 7;
```

Listing 7: Assignment by reference, with comparable C code.

Assigning to an existing variable by reference removes it from existing reference relationships. An assignment by copy to a variable in a reference relationship changes the value of all variables it references.

There is a subtle difference between copying an object and using references. When copying an object between two values, both variables have a separate value, both pointing to the underlying object. By contrast, using references, both variables would share the same value, which points to a single object. In contrast with objects, arrays are deep copied.[6]

2.11 Implicit value creation

Certain assignment statements may have implicit effects, due to PHP's weak-typing. In Listing 8, `$arr` is uninitialized until line 3, when the assignment to `$arr[0]` converts `$arr` into an array. Assignments using object field syntax convert uninitialized values to objects of the class "*stdClass*".

```
1  for ($i = 0; $i < 10; $i++)
2  {
3      $arr[$i] = $i;
4  }
```

Listing 8: An array implicitly created in a loop.

Arrays and objects are also implicitly created using references. An uninitialized variable will be initialized to `NULL` if it is on the RHS of a reference assignment. Referencing fields of arrays or objects will likewise initialize the field. If the array or object itself if not initialized, it too will be created.

2.12 Dynamic code generation

PHP has an `eval` statement which evaluations strings at run-time. The strings may not be known at compile-time, and may even be read from the user. Our analysis expects all code to be statically known, and so we do not deal with this in our analysis. A number of techniques [11, 29] exist to mitigate the pessimistic effects of `eval`s in static analyses.

[6] The PHP reference implementation uses copy-on-write to prevent the cost of these operations, but the semantics are still to copy. [28] presents a problem with this implementation, and a solution. Copy-one-write does not affect our analysis, and so we do not discuss it further.

3. Existing PHP Static Analyses

There has been a significant amount of research into security analysis for PHP.

3.1 WebSSARI

Huang [13] performed the earliest work of which we are aware on the analysis of PHP, using WebSSARI. However, we believe it was a very early prototype which does not attempt to model PHP very well. Xie [30, Section 5.1] discusses the limitations of Huang's work in some detail, noting especially that it is intraprocedural and models only static types. As a result, we do not discuss it further.

3.2 Web vulnerabilities

Xie [30] models a great deal of the simple semantics of PHP5, with the intention of detecting SQL injection vulnerabilities. They model the `extract` function, automatic conversion of some scalar types, uninitialized variables, simple tables, and `include` statements. However, they do not discuss PHP's complex object model or references.

We note a simple misunderstanding of PHP's semantics in Xie's work. In Section 3.1.4, they present the statement:

```
$hash = $_POST;
```

which they claim creates a symbol-table alias of `$POST` in `$hash`. In fact this is a copy of the array in `$_POST`. It is notable that earlier work has misunderstood the semantics of PHP, which demonstrates the difficulty of analysing such a complex and underspecified language.

3.3 Pixy

Pixy [18] provides an alias analysis for PHP 4. They correctly identify that PHP's reference semantics are difficult to model, and that ignoring this feature can lead to errors in program analysis. Their analysis is strongly focused on fixing this flaw.

A contribution of their work is the realization that previous alias analyses [4, 10, 19] for Java, C or C++ are unsuitable for PHP. In particular, PHP's references are mutable, and are not part of the static type of a variable.

Pixy's major contribution is that they model PHP's run-time references between variables in different symbol-tables. Using alias-pairs, they keep track, at each point in the program, of which variables may- or must-alias each other. This includes aliases between variables in the global symbol-table, other symbol-tables, and formal parameters.

There are some limitations with their approach. To begin with, it is rather complicated, with different rules for aliases between two globals, between two formal parameters, between a global variable and a parameter, and between global variables and function-local variables.

More importantly, the alias analysis in Pixy does not go far enough. They do not model that aliases may also exist between variables and members of an array.[7] Unfortunately, this breaks their ***shadow*** model [18, Section 4.4.4] which they use to handle interprocedural analysis.

In our introduction, we highlighted shortcomings in Pixy's lack of type-inference. Pixy inferred that array syntax implied the presence of an array, which was not correct. In addition, they did not propagate the information they did gain, which they refer to as the problem of "*hidden arrays*".

Finally, it is claimed that Pixy does not model a number of PHP 4 features:

- though they are aware of variable-variables, they do not mention a way to model them,
- they cannot model assignments to the `$GLOBALS` variable,
- they do not model functions which affect the local symbol-table (`extract` and `compact`),
- they do not model PHP4's object oriented features.

Since their work is based on PHP4, they also do not model PHP5's new object model.

[7] or an object's fields, but Pixy does not model objects.

3.4 SQL injection attacks

In Wassermann [29] performed a security analysis on PHP programs, aimed at detecting SQL injection vulnerabilities. In [29, Section 5.2], Wassermann details a number of limitations that led to false positives. This included incomplete support for references, and not tracking type conversions among scalar variables. This is similar to the limitations the modelling of arrays in Pixy.

Furthermore, in [29, Section 3.1.1], it is mentioned that SSA form is used. However, since there are a number of features in PHP which can touch the local or global symbol-table, it is difficult to obtain a conservative set of definitions and uses for a function. In fact, as we show in Section 4.7, an advanced alias-analysis is required to build a simple SSA form.

3.5 Unsuitability of alias analysis models for static languages

A traditional alias analysis for Java, C or C++ typically deals with memory locations. For example the "*points-to abstraction*" [10] computes the points-to relation between different stack locations. This is used to model variables which are modified or used by a statement (so-called ***mod-ref analysis*** [19] in C) to allow accurate intraprocedural scalar optimizations.

Java has simpler reference semantics than PHP. All variables and fields are scalars, and some values may be pointers, however pointers to scalar values or to values on the stack are not permitted.

Listing 9 demonstrates references in C++. In it, we see a function ***caller*** which takes a parameter by reference. Its formal parameter `fp` is of type ***int-reference***.

```
void x (int& fp);
```

Listing 9: Example of C++ references.

Throughout the lifetime of `fp`, it is known that:

1. `fp` is a reference,
2. `fp` references a variable which is reachable from outside the scope of `x`.

More importantly, if the reference symbol was omitted, the absence of these features is known. This is not the case in PHP, where the *caller* may declare a parameter to be passed by-reference.

Most existing alias analyses are designed for the semantics of Java, C and C++. They work well because they closely model those semantics. However, since PHP's semantics are only superficially similar to those of Java, C and C++, alias analyses for these static languages are largely not suitable. In particular, existing analyses:

- do not allow for modelling of variable-variables
- are not required to model dynamic effects, such as those relying on type,
- are able to fall back on a static type system [9] in the conservative case,
- may rely on knowing the run-time structures of objects [5].

4. Analysis

Based on the problems with previous analyses, and our experience in compiling PHP in our previous work [3], our analysis has the following major features.

- We model variables as symbol-table fields, making their representation the same as for object fields and array indices. We refer to these as ***names***[8] from here-on-in.

  This allows us to model:

  - all tables using the same elegant abstraction,
  - variable-variables (in the symbol-table) and variable-fields in the same way as array indices,
  - references between each kind of name, modeling many programs that Pixy cannot,
  - the structure of arrays and objects
  - implicit conversions of NULLs to arrays or objects.
- We perform type-inference simultaneously with alias-analysis, constraining each name to a conservative set of types. This allows us to:
  - analyse the program's interprocedural data-flow through polymorphic function calls,
  - model scalar operations, casts, and weak type conversions,
  - track calls to magic methods,
  - prove the absence of object handlers.
- We also propagate literals and constants at the same time.
- Finally, we use the analysis to conservatively model the definitions and uses of names, which is otherwise not straightforward.

The combination of these features is powerful. Only by combining alias analysis, type inference and literal analysis are we able to model types and references of a whole program. Combining them also allows us to model assignments to known array indices, determine if a parameter is passed to a method by-copy or by-reference, resolve many conditional statements, and enable a powerful SSA form on which to base optimizations and further analyses.

Our analysis handles a large majority of PHP's features, with the exception of those listed in Section 4.9. As such, it is the first analysis capable of being used for a conservative analysis of real PHP programs.

4.1 Analysis overview

With these features in mind, we present an overview of our analysis:

- Our analysis performs a symbolic execution of a PHP program, in the same sense as Xie's analysis [30]. We begin at the first statement in the global scope, and perform our analysis one statement at a time. At every statement, we model alias, type, literal, constant and def-use information.
- Our analysis is flow- and context-sensitive, using Pioli's [23] algorithm. We use a worklist algorithm to keep track of the statements in a program.[9] Following Pioli, it is a ***conditional*** analysis, meaning we attempt to resolve branch statements using our literal analysis. This results in an optimistic analysis, which must complete in order for its results to be correct.
- Upon reaching a method invocation, we halt processing of the current method, copy necessary information to the callee (a ***forward-bind*** [4]), and begin processing from the entry block of the invoked method. We build our call-graph lazily, similar to Emami [10] and Burke [4].
- When a method has been fully analysed, we copy information back to the caller (a ***backward-bind*** [4]) and continue the algorithm. If there are multiple possible receivers, we analyse each receiver, and merge their results, before continuing through the caller's worklist.
- After the global scope is fully analysed, we merge the results from each analysed context into a single result for each statement in the program. We then apply our optimization passes, and repeat the analysis until it converges.

4.2 Alias analysis

Our alias analysis is in amalgamation of ideas from previous work. We use a ***points-to-graph***, based on the Emami's "*points-to abstraction*" [10]. We extend this by modelling objects and fields in the manner of Choi's escape analysis [5]. We model references similarly to Pixy [18], but allow references between fields, variables and array values. We use the flow-sensitive model and interprocedural analysis from Pioli [23], itself an extension of the work of Burke [4]. Our context-sensitivity use a call-string approach [24].

Our alias analysis uses a points-to graph to represent the program state. It contains three types of nodes:

- A ***storage node*** represents a table. All arrays, objects and symbol-tables are represented using storage nodes.
- An ***index node*** represents a name, that is, a field of a table. It is used to represent variables, object fields and array indices. Each index node is a child of a single storage node.
- A ***value node*** represents a scalar value. A value node belongs to a single index node, and is not shared between references.

There are three kinds of edges between nodes.

- A ***field edge*** is a directed edge from a storage node to an index node.
- A ***value edge*** is a directed edge from an index node to a storage or value node. If an index node has only one such edge, then the edge's target is its only possible value. Each index node must have at least one outgoing value edge.
- A ***reference edge*** is a bidirectional edge between two reference nodes, indicating that two index nodes reference each other. A reference edge has a ***certainty*** and represents either a ***possible*** reference or a ***definite*** reference (using terms from [10]); that is, two names may-alias, or must-alias.

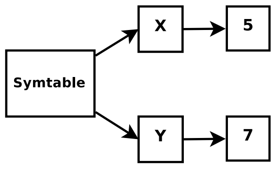
Each array or object allocated in the program is represented by some storage node in the points-to graph. A number of other program constructs require storage nodes:

- a storage node is added for the global symbol-table,
- each class requires a storage node for its static fields,
- a storage node is added for each called function's local symbol-table.

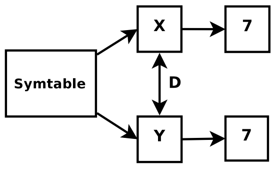
For each storage node in the program, we model an ***UNKNOWN*** field, representing values for unknown names. These are most often used for unknown array indices such as `$arr[$f]`, but they also model variable-variables and variable-fields. Both Jensen [15] and Jang [14] include ***UNKNOWN*** nodes in their Javascript models.

[8] Xie uses the [30] term ***location***, however, we may use multiple names for the same run-time value, which is slightly different the traditional meaning.

[9] Hind [12] process their worklist using topological ordering of the CFG. Similarly, we do not process a statement from the worklist if a statement it post-dominates remains to be processed.



(a) Points-to graph for an assignment by-copy from Listing 6.



(b) Points-to graph for a reference assignment from Listing 7. The reference edge marked **D** is a definite reference edge between `$x` and `$y`.

Figure 3: Points-to graphs, corresponding to the PHP run-time memory models in Figure 2.

For assignments to names which are not statically known, the unknown value is set, as is the value of every other index node in the appropriate storage node(s). Reading from a statically unknown name likewise reads from all possible fields of a storage node. If a name is read which is not present in the points-to graph, the value from the ***UNKNOWN*** node of the appropriate storage node(s) is used instead. As such, an ***UNKNOWN*** node does not represent the values of index nodes which are present in the storage node.

We use call-strings to name storage nodes [24] according to their allocation sites. Each storage node may be concrete (it represents a single run-time structure) or abstract (it may represent multiple run-time structures).

Some index nodes are eligible for strong updates, in which they kill their previous value or reference relationship. A strong update may be performed on an index node i if all of the following conditions are true:

- i's storage node is concrete,
- i is not an ***UNKNOWN*** node,
- i is the only index node referred to by access path of an assignment (see Section 4.6).

At CFG join points, we merge points-to graphs from predecessor basic blocks. We place an edge in the new points-to graph if it exists in either graph. If a reference edge exists in both graphs, then it is ***definite*** if it is ***definite*** in both graphs. In all other cases, it is ***possible***. If a value edge exists in one graph but not the other, we must copy its UNKNOWN value, since it might have been assigned without our knowledge. This also deals with variables and fields which are only initialized in one path, which is common, especially in the presence of loops. The only exception to this rules is that if a storage node only exists in one graph, its index nodes are copied directly, instead of being merged with ***NULL*** values.

4.2.1 Comparison to PHP memory model

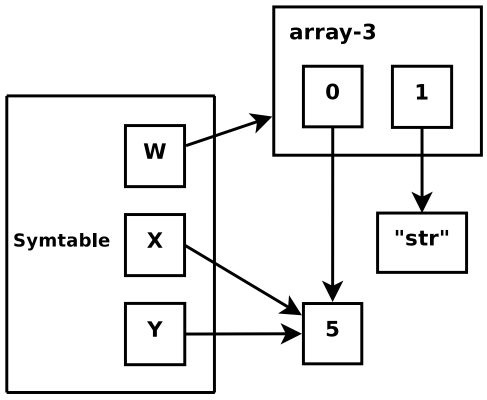
The difference between the PHP run-time memory model and our points-to graphs is demonstrated in Figure 3, corresponding to Listings 6 and 7. The PHP run-time memory model for these listings is shown in Figure 2. For a simple assignment-by-copy, the compile-time and run-time models are largely the same. However, the difference is more visible in an assignment-by-reference. Our points-to graph is required to model representations of the program which cannot occur in practice, so it is slightly more general that the memory representation. Since we model both may-aliases and must-aliases, we are not able to simply use multiple value edges to the same value to model references, as in the PHP run-time memory model. Instead, we are able to model this by using reference edges between aliased index nodes. If we modelled the run-time behaviour by simply sharing value and storage nodes, we would only be able to model may-reference behaviour, resulting in a significant loss of precision. We also use one value node per index node, rather than sharing value nodes between references. If we shared value nodes, then each may-definition (an assignment via a may-reference) would need to be a weak update. Instead, this allows the name being defined to perform a strong update, and only its may-references are weak-updated.

```
$x =  5;
$y =& $x;
$w = array ();
$w[0] =& $x;
$w[1] = "str";
```

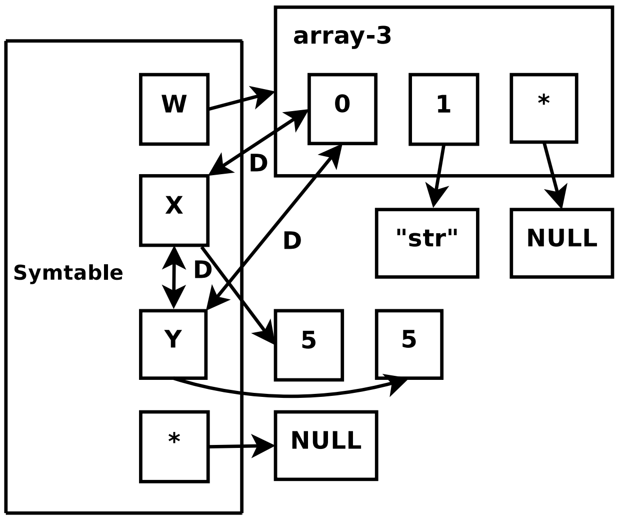
Listing 10: A short program with reference assignments and an array.

For a larger example, Figure 4b shows the points-to graph for Listing 10. The corresponding run-time memory layout is shown in Figure 4a.

At run-time, the symtable has three local variables, `$w`, `$x` and `$y`. `$w` points to an array. `$x` and `$y` alias each other and also alias the zero'th element of `$w`'s array. The array's 1'th value is also shown.



(a) Run-time memory representation



(b) Points-to graph. For simplicity, index nodes are shown within their storage node, instead of using field edges. Edges marked ***D*** are definite reference edges.

Figure 4: Memory layouts for Listing 10.

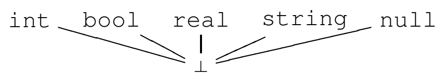


Figure 5: Our literal propagation semi-lattice.

4.3 Literal and type analysis

For each node in the alias analysis, we track information about its literals and types.

We model literal values using a semi-lattice, as shown in Figure 5. At a CFG join point, the meet operation is performed, in which two unequal values merge to $\perp$.[10]

The reason we do not use a lattice is that uninitialized values in PHP evaluate to `NULL`. At a CFG join point, a name with a value (v_1) may meet an uninitialized value (v_2). If we used a lattice, the meet of these two would be v_1 instead of $\perp$ (assuming that v_1 is not ***NULL***. As such, a lattice topped with $\top$ would not be a correct model. We note that Pixy [17] used a lattice, which may lead to incorrectly propagated constants.

The value of the uninitialized name is not guaranteed to be `NULL`, but instead takes its value from the ***UNKNOWN*** value of the appropriate storage node. If the table does not exist, then the uninitialized value is known to be `NULL`.

For each node in the points-to graph we model also a set of types. Names with known values have the type of their value. Otherwise, value nodes are permitted to have any scalar type. Storage nodes are either "*array*"s for arrays and symbol-table, or the single concrete type of an object. Index nodes use the set of the types of all values to which they point. The meet operation for type sets is set union.

For each constant defined in the program, we model its value, if known. If a constant has not been defined, it will evaluate to its name. Constants may only have scalar values, so unknown constants have bounded types. Our analysis can call into the PHP run-time [3], so we have automatic access to constants defined in PHP's standard library, which augments the results of our analysis.

4.4 Termination

We argue informally that our algorithm will terminate, outside the presence of recursion. The semi-lattice is clearly bounded in depth, so literals and constants will converge quickly. The set of types is bounded to the number of types in the program. Since we require all classes to be available at analysis time, this provides a bound on the size of the sets.

Our alias analysis may add at most one storage node per context in the analysis. The number of index nodes a storage node may have is bounded by the number field/variable/array assignments in the program. An assignment to an unknown index node will use ***UNKNOWN***. An assignment to an index node whose name is derived by the literal analysis will converge as the literal analysis does.

4.5 Modelling library functions and operators

PHP has a very large number of built-in and library functions, which are written in C, and therefore not analyzable. The manual documents over 5000 of these function. Since we are required to know the types of a name at all times, we are required to model these functions. We model three aspects:

parameters: Knowing whether a parameter must be passed by reference simplifies analysis and generated code. We retrieve this information automatically using our link to the PHP run-time.

Some functions (typically string or array functions) alter the values of their reference parameters. Modelling these functions precisely can take a some time for the authors of the analysis, so we typically model these conservatively.

A parameter which expects a string may be passed an object with a `__toString` magic method instead, which will be called to evaluate to a string. As such, we model parameters to indicate if they expect a string value. There might also be other magic methods which could be handled in this way, but we have only modelled `__toString` for now.

return values: For most functions it is sufficient to model the type of the return value, usually a small set of scalars. A large number of functions return the value ***false*** in addition to their intended type, to denote an error.

A small minority of functions return some array structure, such as an array of strings. In those cases our model returns an array with the appropriate structure.

purity: There are a large number of functions which are pure (side-effect free). If we know the literal value of all parameters, we may execute this function at compile-time, and return the correct value. Again, this uses our link to the PHP run-time. We use a short timeout for these function calls, but in practice the time taken is negligible.

We also model operators in the same manner: executing them if their operands are known, or modelling their result if it is not. In most cases, the results of simple operands are not simple: the sum of two integers may be a real, the product may be an integer, real or the value false. The semantics of these operations were gleaned from reading the source of the reference implementation — they are not otherwise documented — and we regret not having the space to document them further.

4.6 Access paths

Since the abstraction used for all tables in the program is the same, we are able to map all assignments (lvalues) and expressions (rvalues) to a traversal of the points-to graph. This mapping is referred to as an ***access path*** [19].

An access path consists of a ***storage part*** and an ***index part***, representing the table being indexed, and its key. Consider a simple assignment `$l = $r[$f]`. The lvalue is modelled as:

$$\$l \Longmapsto ST \rightarrow l; \qquad (1)$$

that is, `$l` maps to the local symbol-table indexed by the string "*l*". Both the storage and index parts can themselves be access paths, required for array dereferences:

$$\$r[\$i] \Longmapsto (ST \rightarrow r) \rightarrow (ST \rightarrow i) \qquad (2)$$

In this case, both the array and its index must be first be fetched from the symbol-table. We would expect in general that `$r` will find a storage node corresponding to an array, and the `$i` will find a value node representing an integer or a string. Variable fields are modelled the same way.

Variable-variables are modelled as

$$\$\$x \Longmapsto ST \rightarrow (ST \rightarrow x) \qquad (3)$$

while simpler array assignments are represented as

$$\$r[0] \Longmapsto (ST \rightarrow r) \rightarrow 0 \qquad (4)$$

An access path evaluates to a list of index nodes. If the access path represents an lvalue, the value of the index nodes is being set.

[10] We use $\top$ for uninitialized, and $\perp$ for unknown.

If the path represents an rvalue, the value(s) of the index nodes will be copied or referenced. The algorithm for converting an access path to set of index nodes is straightforward:

- Each access path, where both the storage part `s` and the index part `i` are named, is evaluated. It will correspond with a storage node named `s` with a field named `i`.
- If the resolved access path `p` is part of another path `p'`, then the values of the index nodes represented by `p` are used to resolve `p'`.
- An access path with an unknown index will evaluate to every possible index node of a storage node, including the `UNKNOWN` index node.

4.7 Definitions, uses and SSA form

PHP's syntax provides few clues as to the names which are defined or used in a simple statement. Although aliases exist in more traditional languages like C++, the mutability of PHP references means we cannot make a conservative estimate of PHP's def-use chains [20]. In addition, features like variable-variables, functions which affect the symbol-table of their callers, and object handlers make any syntax based approach meaningless.

While performing the alias-analysis, we record in each statement all names which are defined and used in that statement. We use this to create an SSA form, which would otherwise be impossible to correctly build. The SSA form, based on HSSA [6], creates a platform for further analysis and optimization, which can be built without having to be integrated into the alias analysis. Using this, we build a powerful aggressive dead-code elimination pass [27], [11] which is able to remove reference assignments and dead stores in addition to standard scalar variables. We present results for this analysis in Section 5.1.

The ability to delete reference assignments comes from our def-use model. Each name may, in each statement, be defined, may-defined[12] or used, *by reference or by value*. For an assignment, `$x = $y`:

1. `$x` is defined.
2. `$x`'s reference is used (by the assignment to `$x`).
3. for each alias `$x'` of `$x`, `$x'` is defined. If the alias is ***possible***, it is may-defined instead of defined. In addition, `$x'`'s reference is used.
4. `$y` is used.
5. `$y`'s reference is used.

For an assignment, `$x =& $y`:

1. `$x` is defined.
2. `$x`'s reference is defined (it is not used – `$x` does not maintain its previous reference relationships).
3. `$y` is used.
4. `$y`'s reference is used.

We note that the other names referenced by `$y` do not need to be used. Rather, their use can be surmised by traversing the reference def-use chain starting at `$y`.

4.8 Practical Considerations

Our analysis is performed on a three-address-code (3AC) language. The conversion to 3AC is straight-forward, but the presence of dynamic references leads to some curiosities. Consider the example in Listing 11. Its conversion to 3AC is syntactic, based on the presence of the reference operator (`&`). However, in Listing 12, we cannot convert the array expression to 3AC, as we do not know if the parameter to `foo` is passed by value or reference. As such we must add a ***param-is-ref*** construct, which evaluates at run-time to a boolean indicating whether the n^{th} formal parameter of the function to be called, requires passing by reference. This allows parameters to be converted to 3AC. In general, we can resolve this statically, but there are cases where this is not possible, such as method dispatch where the possible receivers have different signatures.

```
$x =& $a[0][1][2];      $T1 =& $a[0];
                        $T2 =& $T1[1];
                        $x =& $T2[2];
```

Listing 11: An array assignment, by reference, before and after conversion to three-address code.

```
foo ($a[0][1]);         if (param_is_ref (foo, 0)) {
                             $T1 =& $a[0];
                             $T2 =& $T1[1];
                        } else {
                             $T1 =& $a[0];
                             $T2 =& $T1[1];
                        }
                        foo ($T2);
```

Listing 12: A method invocation, by reference, before and after conversion to three-address code.

4.9 Limitations

While we have modelled nearly the entire PHP language, our model is incapable of expressing some parts of PHP. The ***eval*** statement evaluates source code at run-time in the current scope. If we do not know the value of the string beings evaluated, we are unable to know anything about its effects, and our analysis would be forced to return unknown for all names in the program. In practice, we stop the analysis upon discovering an ***eval*** statement. Other analyses for scripting languages have reported the same problem [15, 29]. Furr et al have an interesting partial solution to this problem [11].

We have not modelled magic methods other than `__toString`, but we have instead chosen to prove their absence. We have also chosen to prove the absence of object handlers, instead of modelling them. An object handler is a property of an object, not a class, and so our class based type-inference is slightly too weak to model them. We believe it would not require a large addition, however.

We do not handle the `$_SESSION` variable correctly, modelling it as an array of scalars. In practice, arrays and objects may be persistent from one program iteration to the next. We believe we can handle this by iterating over the other PHP scripts in the application which may be setting this data, or with user annotations. The former is similar to class analyses for C++ [8].

Functions and methods which are called incorrectly are not invoked, but instead return a ***NULL***, after which the caller continues executing. We have not modelled this.

We do not model error handling or exceptions. Both of these are highly dynamic in PHP, and this is the most severe limitation of our analysis.

We do not support dynamic class and method declarations, but these were not intentionally used by program's we have seen (they appeared from ***include*** statements within functions).

We note some areas in which our analysis could be improved. We can often say that a variable must be initialized (if it has a non-NULL value), but we cannot say that it is not initialized.

[11] By dead-code elimination, we mean useless code, not unreachable code (although our analysis never analyses statically unreachable code).

[12] The same could be said of uses, but it isn't useful to do so.

Name	# scripts			SLOC			# statements	
	A	U	P	U	I	P	S	A
RUBBoS	16	18	19	1597	2900	1597	3423	1001
RUBiS	19	20	21	1747	3868	2095	4435	1837
Eve	3	4	8	215	440	907	1473	306
Zend	12	14	14	421	421	421	1398	890
SQLiteAdmin	9	10	16	2791	12005	2915	2583	1212

Figure 6: Characteristics of analysed benchmarks.

5. Discussion

Our primary contributions are our informal description of PHP's behaviour, how it affects conservative program analysis, and how to solve these problems in an static analysis for PHP. Our analysis solves nearly all of the problems we have highlighted. In particular, we have solved many outstanding problems from previous research [18, 29, 30], including extending the alias analysis to model PHP aggregate structures, modelling references between different sorts of names (fields, variables, etc), modelling def-use information suitable for SSA, proving the absence of some complex object-oriented features and modelling implicit and weak type conversions. At the same time, we have presented an alias analysis that extends both work from more static languages [4, 10, 12, 23] and previous scripting language research [18] to fully model PHP's alias behaviour.

5.1 Experimental Evaluation

In this section, we provide an experimental evaluation of our research, analysing a number of benchmarks, and comparing different versions of our algorithm. We implemented our analysis in `phc` [3, 7], our ahead-of-time compiler for PHP. `phc` is open-source,[13] and the analysis algorithm is publicly available. We will soon release a new version of `phc` with the implemented analysis.

We analyse five publicly available PHP programs, including the RUBBoS and RUBiS benchmark suites [2], the Zend benchmark used to benchmark PHP's reference implementation [26], and several programs which were analysed in previous research (Eve Activity Tracker 1.0 and phpSQLiteAdmin 0.2). We note that our analysis is performed once the program is installed, which often includes plugins and language packs. Previous analyses which were non-conservative did not suffer this problem.

In Figure 6 we list statistics from the programs analysed. Each program comprises a number of user-facing PHP scripts. Each user-facing script requires the source of many backing scripts. Since we analyse each script separately, we perform a pre-pass to include all of the backing scripts code automatically.[14]

The ***scripts*** column lists the number of user-facing scripts (U), the number those scripts we analysed[15] (A), and the total number of scripts in the package (P). The ***SLOC*** column lists the number of source lines of code (SLOC) in analysed user-facing scripts (U), the sum of the SLOC of those scripts after the inclusion pre-pass (I), and the sum of the SLOC of each file in the package (P). The ***statements*** column shows the number of static statements in the program (S), and the number of statements in the program which we traversed during analysis(A). All further figures are aggregated over each analysed scripts in a test package.

We analyse each program four times, varying the context- and flow-sensitivity. Our context-sensitive version uses an infinite call-string, our insensitive version uses a call-string length of one (that is, only a single statement ID is used in the name of the storage node created). Our flow-sensitive version is described in the previous section; our flow-insensitive version is simply the flow-sensitive version, except we always perform weak updates instead of strong updates. The ***# statements*** column in Figure 6 lists the static number of three-address-code [20] statements in the program, and the number of statements analysed in the flow-sensitive version of our analysis (first context-insensitive, then context-sensitive). We do not present the number of statements analysed in the flow-insensitive version, since it is only an approximation of a true flow-insensitive version.

[13] Available from `http://www.phpcompiler.org`.

[14] Some programs were changed to make this straightforward, by changing PHP constants to hard-coded strings. This did not change any program semantics.

[15] We skipped some scripts due to minor bugs.

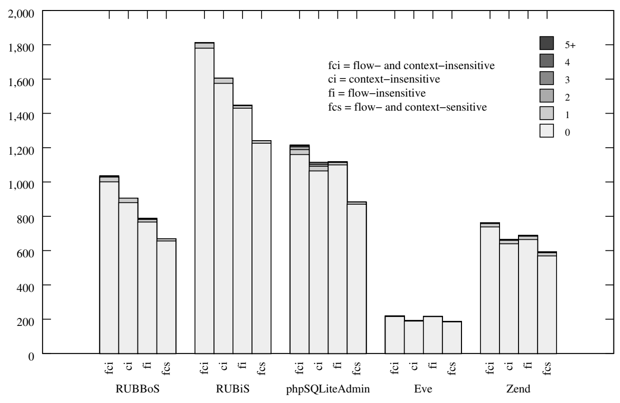


Figure 7: Peak references per variable in the analysed benchmarks, parametrised by flow- and context-sensitivity.

We ran our analysis, followed by a number of optimizations including constant propagation and folding, dead code elimination, and removal of calls to pure or empty functions. We modelled 220 simple built-in PHP functions which required only the details in Section 4.5, and 56 more complex functions.

Figure 7 presents the variables in a program, along with the number of times they are aliased. Unaliased variables may frequently be optimized much more heavily than aliased variables, by putting them into registers, or performing the ***scalar replacement of aggregates*** optimization [20]. The number of variables is lower in the more precise versions of the program, principally due to dead code elimination. For each variable in a function, we count the peak number of aliases it has over its lifetime.

We see that the vast majority of variables in the program are unaliased, and that it is largely invariant in the flow- and context-sensitivity. It is clear that context-insensitivity increases aliasing slightly, but not by a significant amount. We see also that flow- and context-sensitivity generally decrease the number of variables in a program with context-sensitivity being the more significant factor. This is probably because more accurate context-sensitive analysis leads to fewer abstract storage nodes, allowing for better constant propagation.

Figure 8 shows a distribution of the number of types of variables in a program. Statically knowing that a variable has only a single type allows bug-finding tools to eliminate false positives, and allows compilers to generate better code. A low number of static types also has significant benefits for IDEs and code browsers, which has been lacking for scripting languages.

For each variable in a function, we count the number of types it has over its lifetime. This is performed after constant propagation, and so does not include statements which can be optimized to take a constant. Our results show that over 60% of variables in PHP programs have a single known type, and 90% of variables have two possible types or fewer, using flow- and context-sensitive analysis.

In our experience, many variable which have greater than one type have a possible `NULL` value. This is particularly prevalent in loops, where variables are often uninitialized on the first loop exe-

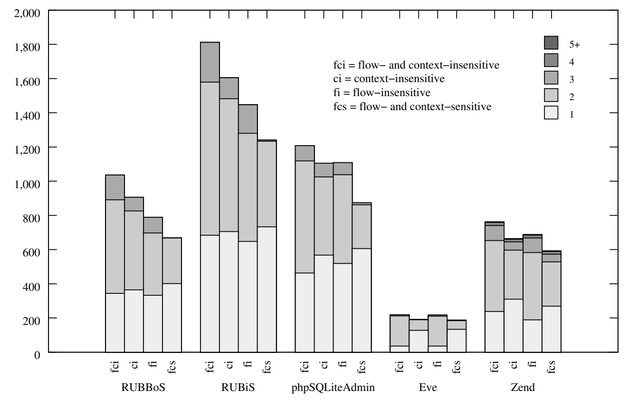


Figure 8: Peak types per variable in the analysed benchmarks, parametrised by flow- and context-sensitivity.

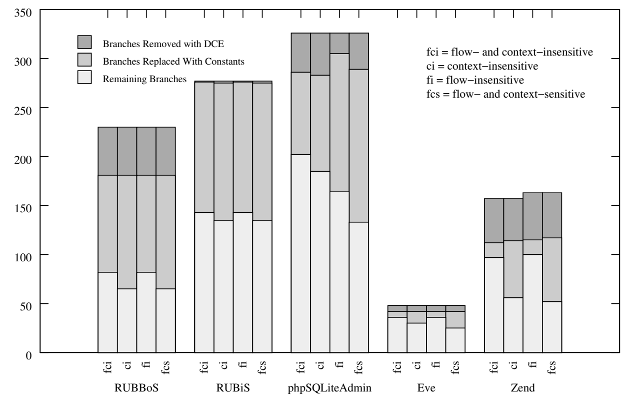


Figure 9: Branches removed in the analysed benchmarks, parametrised by flow- and context-sensitivity.

cution, and have static types for the rest of the loop's execution. We speculate that unrolling the first loop iteration would be beneficial in an optimizing compiler, which we intend to investigate in future work. In addition, arithmetic operations and calls to built-in functions rarely return a single type, with the former usually resulting in the set ***(real, int)***, and the latter possibly returning `false` as well as its intended value in many cases.

We believe these three cases account for a majority of variables which have more than one type, and speculate that the program is much more statically typed than our current results show. This lends some credence to Huang's [13] simple modelling of PHP programs using static types, which it seems is acceptable in a pinch.

Figure 9 shows the number of branch statements in our program, and the number that we are able to remove with optimization. The peak in each column shows the sum of the number of branches at the start of the program, and those created by small optimizations which we have not discussed.[16] ***Branches after optimization*** shows the number of branches which remain after the analysis has converged and all optimizations have been performed. The removed branches are split into those removed by dead-code elimination (***Branches removed by DCE***), and those where the branch direction is known (***Branches replaced with constants***). We can see that flow-sensitive analysis gives better results, significantly better in some cases, as might be expected. We note that our intermediate representation does not typically create opportunities for this optimization, except in the case of the ***param-by-ref*** construct,

[16] These optimizations are the reason that the *Zend* results are not flat.

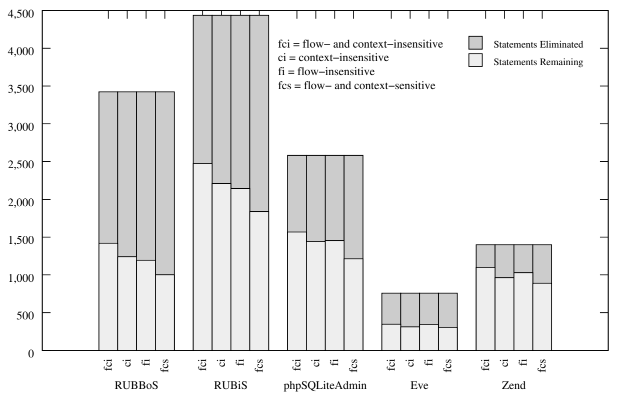


Figure 10: Dead code eliminated in the analysed benchmarks, parametrised by flow- and context-sensitivity.

which should be equally well known in both the flow-sensitive and -insensitive cases.

Finally, we present the number of statements eliminated by dead-code elimination in Figure 10. Our intermediate representation creates a very large number of opportunities for dead code. Constants and literals are moved into variables during the creation of 3AC, and ***param-is-ref*** statements are added. We see that a great number of these statements are removed through analysis, in one case over 70%.

A clear result of our analysis is the effectiveness of static analysis for scripting languages. Though we have seen that the analysis is typically much more difficult to perform than for more static languages, the results are excellent, with a very large number of statically typed variables, and a very low number of aliased variables. IDEs in particular can gain a great deal of information from this analysis. In addition, the optimization opportunities are clearly very good, with large quantities of code removed, a majority of variables which are unaliased, and a large number of variables which may only be a single type over their life time.

We have seen many claims that just-in-time (JIT) compilers are the only way forward for compilers for dynamic scripting languages. We believe that these preliminary results indicate otherwise.

6. Related work

Our analysis is similar to that of Jensen et al [15]. They perform a type analysis for Javascript, which has many of the same difficulties as PHP, and come up with many similar solutions. Their lattice models both types and literal values, they model default values for missing properties, and they model the structure of objects using a pointer analysis.

The differences between the two analyses stem largely from the differences between the languages. While PHP has a static inheritance hierarchy, Javascript's is dynamic, and their type analysis attempts to model that finely. By contrast, Javascript does not have PHP-like references. This is visible from their work: they discuss points-to analysis only in passing, as existing solutions seem to satisfy their need. In contrast, dynamic references are the main static analysis challenge in PHP, and we use type inference and constant propagation to support our alias analysis and improve its precision.

Jang [14] presents a points-to analysis for Javascript. Its primary contribution is that it uses the same mechanism for array and property accesses. They do not perform type inference, and their pointer analysis is covered by Jensen's work.

Although scripting languages are a departure from traditional languages, they share many similarities with languages which have been analysed before. The SELF project in particular has performed important research on type-inference [1].

7. Conclusion

Scripting languages have become some of the most widely used programming languages, particularly for web development. The dynamic features of scripting languages make static analysis very challenging, particularly for PHP which has no documented semantics outside the source code of its reference implementation. However, we have extensively documented PHP's run-time behaviour, how it affect program analysis, and in particular its difficult-to-analyse features, for the first time.

We have developed a static analysis model for PHP that can deal with dynamic language features such duck-typing, dynamic and weak typing, overloading of simple operations, implicit object and array creation and run-time aliasing. The main focus of our work is alias analysis, but we show how type inference and constant propagation must be used to perform the analysis effectively. We also show how SSA form cannot be used without the presence of a powerful alias analysis.

Our analysis has been implemented in the `phc` ahead-of-time compiler for PHP, and used to analyse a number of real PHP programs totalling 19684 lines of source code. We have found that our analysis is able to determine that almost all variables are unaliased. We are also able to statically determine the dynamic type of 60% of variables in our test programs. Finally, we have provided comparisons of context- and flow-sensitive and -insensitive variants of our algorithm and find that both context- and flow-sensitivity are valuable in increasing the accuracy of the analysis.

Acknowledgements

The authors are indebted to Irish Research Council for Science, Engineering and Technology funded by the National Development Plan, whose funding made this work possible. Jimmy Cleary worked hard on the experiments, and this paper would not have been complete without his help and dedication. ... provided valuable insights on a draft of this paper.

put in final list of acknowledgements

References

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26. Springer-Verlag, 1995.

[2] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *In 5th IEEE Workshop on Workload Characterization*, pages 3–13, 2002.

[3] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1916–1923. ACM, 2009.

[4] M. G. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *LCPC*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer, 1994.

[5] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[6] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267. Springer-Verlag, 1996.

[7] E. de Vries and J. Gilbert. Design and implementation of a PHP compiler front-end. Dept. of Computer Science Technical Report TR-2007-47, Trinity College Dublin, 2007.

[8] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer-Verlag, 1995.

[9] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *PLDI*, pages 106–117, 1998.

[10] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.

[11] M. Furr, J. hoon (David) An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, 2009.

[12] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[13] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW*, pages 40–52. ACM, 2004.

[14] D. Jang and K.-M. Choe. Points-to analysis for javascript. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1930–1937. ACM, 2009.

[15] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[16] N. Jovanovic. *Web Application Security*. PhD thesis, Technischen Universitat Wien, 2007.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. Technical report, Vienna University of Technology, 2006.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36. ACM, 2006.

[19] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.

[20] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[21] K. Ostermann. Nominal and structural subtyping in component-based programming. *Journal of Object Technology*, 7(1):121–145, 2008.

[22] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[23] A. Pioli, M. Burke, and M. Hind. Conditional pointer aliasing and constant propagation. Master's thesis, SUNY, New Paltz, 1999.

[24] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. Prentice Hall International, 1981.

[25] The PHP Documentation Group. *PHP Manual*, 1997-2009. `http://www.php.net/manual/`.

[26] The PHP Group. Zend benchmark, 2007. `http://cvs.php.net/viewvc.cgi/ZendEngine2/bench.php?view=co`.

[27] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2007.

[28] A. Tozawa, M. Tatsubori, T. Onodera, and Y. Minamide. Copy-on-write in the php language. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 200–212. ACM, 2009.

[29] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41. ACM Press, 2007.

[30] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium,*, pages 179–192, July 2006.