Precise Alias Analysis for Static Detection of Web Application Vulnerabilities

Nenad Jovanovic Christopher Kruegel Engin Kirda

Secure Systems Lab
Technical University of Vienna
{enji,chris,ek}@seclab.tuwien.ac.at

Abstract

The number and the importance of web applications have increased rapidly over the last years. At the same time, the quantity and impact of security vulnerabilities in such applications have grown as well. Since manual code reviews are time-consuming, error-prone and costly, the need for automated solutions has become evident.

In this paper, we address the problem of vulnerable web applications by means of static source code analysis. To this end, we present a novel, precise alias analysis targeted at the unique reference semantics commonly found in scripting languages. Moreover, we enhance the quality and quantity of the generated vulnerability reports by employing a novel, iterative two-phase algorithm for fast and precise resolution of file inclusions.

We integrated the presented concepts into Pixy [14], a high-precision static analysis tool aimed at detecting cross-site scripting vulnerabilities in PHP scripts. To demonstrate the effectiveness of our techniques, we analyzed three web applications and discovered 106 vulnerabilities. Both the high analysis speed as well as the low number of generated false positives show that our techniques can be used for conducting effective security audits.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.2.4 [*Software Engineering*]: Software/Program Verification—Validation

General Terms Verification, Security, Languages

Keywords alias analysis, data flow analysis, static analysis, program analysis, web application security, scripting languages security, cross-site scripting, PHP

1. Introduction

Web applications have become one of the most important communication channels between various kinds of service providers and clients on the Internet. Along with the increased importance of web applications, the negative impact of security flaws in such applications has grown as well. Vulnerabilities that may lead to the compromise of sensitive information are being reported continuously, and the costs of the resulting damages are increasing. The main reasons for this phenomenon are time and financial constraints, limited programming skills, and lack of security awareness on part of the developers.

The existing approaches for mitigating threats to web applications can be divided into client-side and server-side solutions. The only client-side tool known to the authors is Noxes [16], an application-level firewall offering protection in case of suspected *cross-site scripting* (XSS) attacks that attempt to steal user credentials. Server-side solutions have the advantage of being able to discover a larger range of vulnerabilities, and the benefit of a security flaw fixed by the service provider is instantly propagated to all its clients. These server-side techniques can be further classified into dynamic and static approaches. Dynamic tools (e.g., [11, 23, 26], and Perl's taint mode [30]) try to detect attacks while executing the audited program, whereas static analyzers ([12, 13, 14, 20, 21, 33]) scan the entire web application's source code for vulnerabilities before it is deployed.

In a previous paper, we presented Pixy [14, 15], the first open source tool for statically detecting taint-style vulnerabilities (in particular, XSS vulnerabilities) in PHP 4 [25] code. Pixy features a high-precision data flow analysis engine that is flow-sensitive, interprocedural, and context-sensitive and performs alias analysis, literal analysis, and taint analysis. These characteristics enabled it to generate more comprehensive and precise results than those provided by other approaches [12, 13, 21]. In this paper, we further improve our work in the following key points:

- We present a novel, precise alias analysis targeted at the unique reference semantics commonly found in scripting languages. Without a preceding alias analysis, taint analysis would generate false positives as well as false negatives in conjunction with aliases. In contrast to our previous approach, our new alias analysis generates precise results even for conceptually difficult aliasing problems. Our decision to develop a new alias analysis is based on the belief that the straightforward adaptability of existing solutions to this problem domain seems questionable, since they are targeted at non-scripting languages such as C and Java. This belief is backed by the observation that we were the first to analyze aliases in the context of scripting languages.
- We enhance the quality and quantity of the generated vulnerability reports as well as our tool's usability by integrating a novel, iterative two-phase algorithm for fast and precise resolution of file inclusions. In C, include statements only contain static file names and thus, can be resolved easily. In PHP, however, include statements can be composed of arbitrary expressions, requiring more sophisticated resolution techniques.
- We present empirical results for demonstrating that our tool can be used to detect XSS vulnerabilities in real-world programs.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.

```
echo "Here is what you wrote: " . $_GET['content'];
```

Figure 1. Very simple PHP script vulnerable to XSS.

The analysis process is fast, completely automatic, and produces a low false positive rate.

The rest of the paper is structured as follows. Section 2 introduces the general class of vulnerabilities that Pixy aims to detect. In Section 3, we present an overview of our analysis infrastructure. Section 4 provides the details of our alias analysis, and Section 5 explains the workings of the include resolution algorithm. A summary of our empirical results is presented in Section 6. After a discussion of related work in Section 7, Section 8 briefly concludes.

2. Taint-Style Vulnerabilities and XSS Attacks

The analysis infrastructure that we are enhancing in this paper is targeted at the detection of taint-style vulnerabilities. *Tainted* data denotes data that originates from potentially malicious users and thus, can cause security problems at vulnerable points in the program (called *sensitive sinks*). Tainted data may enter the program at specific places, and can spread across the program via assignments and similar constructs. Using a set of suitable operations, tainted data can be *untainted* (*sanitized*), removing its harmful properties. Many important types of vulnerabilities (e.g., cross-site scripting, SQL injection, or script injection) can be seen as instances of this general class of *taint-style vulnerabilities*. An overview of these vulnerabilities is given by Livshits and Lam in [20].

One of the main purposes of XSS attacks [5] is to steal the credentials (e.g., the cookie) of an authenticated user. Every web request that contains an authentication cookie is treated by the server as a request of the corresponding user as long as she does not explicitly log out. Thus, everyone who manages to steal the cookie is able to impersonate its owner for the current session. The browser automatically sends a cookie only to the web site that created it, but with JavaScript, a cookie can be sent to arbitrary locations. Fortunately, the access rights of JavaScript programs are restricted by the *sandbox model*. That is, a JavaScript program has access only to cookies that belong to the site from which the code originated.

XSS attacks circumvent the sandbox model by injecting malicious JavaScript into the output of vulnerable applications. In this case, the malicious code appears to originate from the trusted site and thus, has complete access to all (sensitive) data related to this site. For example, consider the simple PHP script in Figure 1, where a user's posting to a message board is displayed after submitting it. The posting's content is retrieved from a GET parameter. Therefore, it can also be supplied in a specifically crafted URL such as the following, which results in the user's cookie being sent to "evilserver.com":

```
http://vulnerable.com/post.php?
content=<script>document.location=
'evilserver.com/steal.php?'+document.cookie</script>
```

All that the attacker has to do is to trick a user into clicking this link, for example, by sending it to the victim via email.

In general, an XSS vulnerability is present in a web application if malicious content (e.g., JavaScript) received by the application is not properly stripped from the output sent back to a user. When speaking in terms of the sketched class of taint-style vulnerabilities, XSS can be roughly described by the following properties:

- Entry Points into the program: GET, POST and COOKIE arrays.
- Sanitization Routines: htmlentities(), htmlspecialchars(), and type casts that destroy potentially malicious characters or transform them into harmless ones (such as casts to integer).
- Sensitive Sinks: All routines that display data on the screen, such as echo(), print() and printf().

3. Analysis Overview

We built our alias analysis into Pixy [14], a tool that performs data flow analysis on PHP code to detect XSS vulnerabilities. Data flow analysis is a well-understood topic in computer science and has been used in compiler optimizations for decades ([1, 22, 24]). In a general sense, the purpose of data flow analysis is to statically compute certain information for every single program point (or for coarser units such as functions). For instance, the classical *constant analysis*[1] computes, for each program point, the literal values that variables may hold.

Data flow analysis operates on the control flow graph (CFG) of a program. Hence, a parse tree is constructed from a PHP input file using the Java lexical analyzer JFlex and the Java parser Cup (JCup). The parse tree is then transformed into a linearized form resembling three-address code with basic blocks [1], and stored as a separate control flow graph for each encountered function. Then, file inclusions are resolved using an iterative, two-stage preprocessing step described in Section 5. This preprocessing step employs literal analysis for computing the names of files referenced by non-literal include statements. Note that file inclusion is a transitive process. That is, include statements found in files that were previously included are resolved as well. Afterwards, the alias analysis discussed in Section 4 computes alias relationships for each variable at every program point. Finally, this information is utilized by a taint analysis for determining the taint values of variables and reporting tainted variables that enter sensitive sinks.

4. Alias Analysis

For static detection of vulnerabilities, precise results can only be achieved by considering possible alias relationships between variables. Two or more variables are *aliases* at a certain program point if their values are stored at the same memory location. Two variables are *must-aliases* if they are aliases regardless of the actual path that is taken by the program during run-time. If these variables are aliases only for some program paths, while not for others, they are called *may-aliases*. We will give a short introduction to aliases in PHP to demonstrate why alias information is required for precise results, and to highlight the differences between PHP aliases and pointers in other programming languages. After this problem definition, we specify the workings of our alias analysis, which is responsible for computing the desired information.

4.1 Aliases in PHP

In PHP, aliases between variables can be introduced by using the *reference operator* "&". This operator can be applied directly in assignments, or in combination with formal and actual function parameters to perform a call-by-reference. Figure 2 shows a simple example for creating an alias relationship between variables $a and $b (on Line 2). This figure also demonstrates why taint analysis requires access to alias information. Without this information, taint analysis would not be able to decide that the assignment on Line 3 does not only affect $a, but also the aliased variable $b. As a result, we would miss the fact that $b eventually holds a tainted value, which leads to the XSS vulnerability on Line 4. Analogously, the lack of aliasing information can cause false positives.

[1] Note that we will use the name "literal analysis" instead of the classical term "constant analysis" in order to prevent confusion with PHP's constants.

```
1: $b = 'nice';   // $b: untainted
2: $a =& $b;      // $a and $b: untainted
3: $a = $evil;    // $a and $b: tainted
4: echo $b;       // XSS vulnerability
```

Figure 2. Simple aliasing in PHP.

```
1: $x1 = 1;
2: $x2 = 2;
3: a(&$x1);
4: echo $x1;   // $x1 is still '1'
5:
6: function a(&$p) {
7:   $p =& $GLOBALS['x2'];
8: }
```

Figure 3. References in contrast to pointers.

In the past, extensive work has been devoted to the area of alias analysis (e.g., [2, 6, 18, 29, 32], to mention only a few). An overview of existing solutions and open issues is given by Hind in [10]. However, an important aspect to note is that the semantics of references are fundamentally different from those of pointers in languages such as C. The PHP Manual [25] devotes a whole chapter to explaining references and highlighting the differences to C pointers. In essence, while C pointers are special variables that contain memory addresses, PHP references are *symbol table aliases* [25] that do not directly address memory locations. Besides, PHP does not provide a separate data type for references. Instead, *all* variables are references by nature, even those containing only scalar values. Another difference, which occurs in combination with parameter passing, is illustrated by Figure 3. When entering function "a" on Line 6, the formal parameter $p has been aliased with the actual parameter $x1. However, since $x1 and $p are now only symbol table aliases, the reference assignment on Line 7 only re-references $p, leaving $x1 unmodified. In C, passing and modifying a pointer in this way would make the pointer corresponding to $x1 point to $x2 after returning from the function call on Line 3. To the best of our knowledge, these issues have only been superficially addressed in one of our previous works [15] so far. Moreover, PHP references are mutable, as opposed to references in C++. Finally, as mentioned by Xie and Aiken [33], the fact that PHP is a scripting language leads to further problems for static analysis, such as implicit and dynamic type information, or the lack of explicit variable declarations. Liu et al. [19] briefly mention to have applied existing pointer analysis algorithms to Python programs, unfortunately without providing further details.

4.2 Intraprocedural Alias Analysis

Figure 4 shows a program snippet annotated with alias information that is valid after the execution of the corresponding code line. In this figure (and in the following ones), we represent must-alias ("u") and may-alias ("a") information separately. At the beginning of the program on Line 1, there exist no aliases yet. After the reference assignment on Line 2, variables $a and $b are aliases. We encode this fact by adding a new *must-alias group* to the must-alias information. Must-alias groups are unordered and disjoint sets of variables that are must-aliases. On Line 4, a second group is created after redirecting $c to $d. This new group is extended by variable $e as result of the statement on Line 5. Finally, we have to merge the information entering from two different paths after the if-construct on Line 7. Intuitively, it is clear that all must-aliases created inside the if-construct must be converted into may-aliases.

```
1: skip;          // u{} a{}
2: $a =& $b;      // u{(a,b)} a{}
3: if (...) {
4:   $c =& $d;   // u{(a,b) (c,d)} a{}
5:   $e =& $d;   // u{(a,b) (c,d,e)} a{}
6: }
7: skip;          // u{(a,b)} a{(c,d) (c,e) (d,e)}
```

Figure 4. Intraprocedural analysis information.

Instead of using sets of variables, we encode may-aliases by means of unordered variable pairs. Hence, the must-alias group (c,d,e) is split into the three may-alias pairs (c,d), (c,e), and (d,e). The reason for this asymmetric encoding of must-alias and may-alias information is that it simplifies the algorithms necessary for interprocedural analysis (given in Appendix A). Figure 13 in Appendix A shows the combination operator algorithm that is used for merging alias information at the meeting point of different program paths (based on the construction of strongly connected components). Note that this combination operator does not simply compute must-aliases through intersection and may-alias through union (although these steps are performed as parts of the algorithm). For instance, using such a straightforward procedure to combine the information from Lines 2 and 5 of Figure 4 would result in empty may-alias information, which deviates from the correct result shown on Line 7.

The separate tracking of must-alias and may-alias information (instead of using only may-alias information) is motivated by the resulting precision gain. Consider the case where variables $a and $b are must-aliases and tainted. When encountering an operation that untaints $a, our analysis is able to correctly untaint $b as well. If the analysis only possesses may-alias information, it would have to make a conservative decision and leave $b tainted.

4.3 Interprocedural PHP Concepts

Before going into the details of our interprocedural alias analysis, we will give a brief overview of the PHP concepts necessary for understanding the following sections. In terms of scoping, there are two types of variables in PHP: local variables, which appear in the local scope of functions, and global variables, which are located in the global scope (i.e., outside every function). Note that formal function parameters belong to the class of local variables. From inside functions, global variables can be accessed in two ways. The first method is using the "global" keyword. A statement such as "global $x" has the effect that the local variable $x is aliased with the global variable $x. The other way is to access global variables directly via the special "$GLOBALS" array, which is visible at every point in the program. Using this array, global variables can even be re-referenced from inside functions, whereas the "global" keyword does not offer this possibility.

4.4 Interprocedural Alias Analysis

The main problem arising with interprocedural analysis is the handling of recursive function calls. Every instance of a called function contains its own copies of its local variables (*variable incarnations*). In most cases, it is not possible to decide statically how deep recursive call chains can become since the depth may depend on dynamic aspects, such as values originating from databases, or user input. Hence, static analysis would be faced with an infinite number of variable incarnations. Since this would mean that the underlying lattice would not satisfy the ascending chain condition [24] (i.e., it would have an infinite height), the analysis would not terminate in such cases. The solution to this problem is the following:

> *Inside functions, the analysis only tracks information about global variables and its own local variable incarnations.*

In the global scope, only global variables are considered. This vital rule leads to a finite number of variables during the analysis and forms the basis for the rest of the paper.

When encountering a function call during the analysis, the following two questions arise:

1. What alias information has to be propagated into the callee?
2. What alias information is valid after control flow returns to the caller?

We will now give an overview of the answers to these questions. A more detailed treatment will be presented afterwards. From the callee's point of view, the analysis has to provide the following information:

- Aliases between global variables.
- Aliases between the callee's formal parameters.
- Aliases between global variables and the callee's formal parameters.

From the caller's point of view, the following information has to be obtained after the function returned:

- Aliases between global variables.
- Aliases between global variables and the caller's local variables.

Note that the aliases between the caller's local variables cannot be modified by the callee. Similarly, the aliases between the callee's local variables are always the same on function entry. In the following sections, we will discuss each of the above issues in detail, ordered by increasing complexity of the necessary concepts. The detailed algorithms can be found in Appendix A.

4.4.1 Aliases between Global Variables

The alias relationships between global variables are important for both the caller and the callee. On the one hand, the callee must know about how global variables are aliased at the time the function call is performed. On the other hand, the caller must be informed about how the global aliasing information has been modified by the callee. These aspects can be treated in a straightforward way, similar to the method applied by Sharir and Pnueli in their classic treatment of interprocedural analysis [28]. An example is given in Figure 5. In this figure, we extend our notation by prefixing variable names with the name of the containing function. Global variables are considered to be contained in the special "main" function, abbreviated with "m". When calling function "a" on Line 2, there is no aliasing at all. This empty alias information is propagated into the function. From the function's entry until the call to "b" on Line 9, we simply apply our intraprocedural techniques. As mentioned above, each function only tracks information about global variables and its own local variables. Therefore, the information about the local variables of "a" is removed prior to propagation into "b". The information about global variables, however, is propagated as it is. Inside function "b", the global aliases are modified by the statement on Line 15. On Line 10, this modified information is returned to function "a", which also restores the alias information for its own local variables. May-aliases between global variables, which have not occurred in this example, are treated analogously.

4.4.2 Aliases between the Callee's Formal Parameters

Aliases between *formal* parameters appear when there exists an alias relationship between the corresponding *actual* call-by-reference parameters. For instance, function "b" in Figure 6 has two call-by-reference parameters, $bp1 and $bp2. The corresponding actual parameters are $a1 and $a2, which are must-aliases at the time of the call to function "b". As a result, the formal parameters $bp1 and $bp2 are must-aliases on function entry.

```
01: skip;  // u{} a{}
02: a();
03: skip;  // u{(m.x1, m.x2, m.x3)} a{}
04:
05: function a() {  // u{} a{}
06:   $a1 =& $a2;   // u{(a.a1,a.a2)} a{}
07:   $GLOBALS['x1'] =& $GLOBALS['x2'];
08:   skip;      // u{(a.a1,a.a2) (m.x1, m.x2)} a{}
09:   b();
10:   skip;      // u{(a.a1,a.a2)
11:              //   (m.x1, m.x2, m.x3)} a{}
12: }
13:
14: function b() {  // u{(m.x1, m.x2)} a{}
15:   $GLOBALS['x3'] =& $GLOBALS['x1'];
16:   skip;  // u{(m.x1, m.x2, m.x3)} a{}
17: }
```

Figure 5. Aliases between global variables.

```
01: a();
02:
03: function a() {     // u{} a{}
04:   $a1 =& $a2;      // u{(a.a1, a.a2)} a{}
05:   b(&$a1, &$a2);
06: }
07:
08: function b(&$bp1, &$bp2) {
09:   skip;   // u{(b.bp1, b.bp2)} a{}
10: }
```

Figure 6. Must-aliases between formal parameters.

For the treatment of may-aliases between formal parameters, additional considerations are necessary. First, recalling that may-alias pairs are unordered, we can identify three types of may-alias pairs that can exist at the time of a function call: (local, local), (global, global), and (local, global). Next, we can distinguish several cases depending on how many elements of a may-alias pair are used as actual call-by-reference parameter (either one or both). Of course, if no element of a may-alias pair is used as parameter, it cannot induce aliases between formal parameters. Table 1 provides an overview of all possible cases and the may-alias pairs resulting for the callee. The table shows the may-aliases between the formal parameters of a function with signature b(&bp1, &bp2) that result from different calls to this function (given in the first column) and different may-aliases at the time of the function call (given by the second column, labeled with "Entering may-aliases"). An example for the case in the second row of Table 1 is shown in Figure 7. Here, the may-alias pair ($a1, $a2), which consists of two local variables, reaches the call to function "b" on Line 8. Both of these local variables are used as actual call-by-reference parameters. Hence, this initially results in three may-alias pairs: ($bp1, $bp2), ($bp1, $a2), and ($bp2, $a1). The last two pairs are not propagated to the callee, since they contain local variables of the caller. Figure 14 in Appendix A shows the exact algorithm that has been applied here.

4.4.3 Aliases between Global Variables and the Callee's Formal Parameters

For detecting aliases between global variables and the callee's formal parameters, we have to consider the following cases for the actual call-by-reference parameter:

- The parameter is a must-alias of a global variable.

Function call	Entering may-aliases	Resulting relevant may-aliases	Resulting irrelevant may-aliases
b(&$local_1, –)	(local_1, local_2)	none	(bp1, local_2)
b(&$local_1, &$local_2)	(local_1, local_2)	(bp1, bp2)	(bp1, local_2), (bp2, local_1)
b(&$global_1, –)	(global_1, global_2)	(bp1, global_2)	none
b(&$global_1, &$global_2)	(global_1, global_2)	(bp1, global_2), (bp2, global_1), (bp1, bp2)	none
b(&$local, –)	(local, global)	(bp1, global)	none
b(&$global, –)	(local, global)	none	(bp1, local)
b(&$local, &$global)	(local, global)	(bp1, bp2), (bp1, global)	(bp2, local)

Table 1. May-aliases between formal parameters resulting from calls to a function with signature b(&bp1, &bp2)

```
01: a();
02:
03: function a() {      // u{} a{}
04:   if (...) {
05:     $a1 =& $a2;     // u{(a.a1,a.a2)} a{}
06:   }
07:   skip;             // u{} a{(a.a1,a.a2)}
08:   b(&$a1, &$a2);
09: }
10:
11: function b(&$bp1, &$bp2) {
12:   skip;       // u{} a{(b.bp1,b.bp2)}
13: }
```

Figure 7. May-aliases between formal parameters.

```
01: a();
02:
03: function a() {             // u{} a{}
04:   $a1 =& $GLOBALS['x1'];  // u{(a.a1,m.x1)} a{}
05:   b(&$a1);
06: }
07:
08: function b(&$bp1) {    // u{(m.x1,b.bp1)} a{}
09:   skip;
10: }
```

Figure 8. May-aliases between formal parameters and global variables.

- It is a global variable (and hence, a trivial must-alias of a global variable).
- It is a may-alias of a global variable.

Fortunately, these cases are quite simple and can be handled with the same means as those that have been applied in the previous section. Figure 8 shows an example for the first case. At the call to function "b" on Line 5, variable $a is a must-alias of the global variable $x1. Since $a is used as actual call-by-reference parameter, this means that the formal parameter $bp1 becomes a must-alias of $x1 on function entry.

4.4.4 Aliases between Global Variables and the Caller's Local Variables

As mentioned previously, the aliases between local variables of a caller cannot be changed by a callee. However, the aliases between the caller's local variables and global variables can be modified by the callee in the following ways:

1. If a local variable is aliased with a global variable at the time of the function call:
   (a) Other global variables can be redirected to this global variable, and hence, to the local variable.
   (b) This global variable can be redirected to something else, and hence, away from the local variable.
2. If a local variable is aliased with a formal parameter through call-by-reference:
   (a) Global variables can be redirected to this formal parameter, and hence, to the local variable.

Note that each of these cases implies a number of subcases depending on whether must- or may-aliasing is performed. Our basic rule for interprocedural analyses forbids the propagation of aliasing information about local variables to other functions. Hence, another mechanism is necessary to be able to collect information about changes of aliasing relations between global variables and local variables. For this purpose, we will present the notion of *shadow variables*.

Shadow Variables Our analysis uses two types of special variables for solving the problem mentioned above. The first type, called *formal-shadows* (or *f-shadows*), are introduced at the beginning of every function. There is one f-shadow for each formal parameter of a function, and each f-shadow is aliased with its corresponding formal parameter at the beginning of this function. For instance, consider the function with signature "a($ap1, $ap2)". The analysis introduces the f-shadows $ap1_fs and $ap2_fs at the beginning of the function, and aliases them with their formal parameters. Therefore, $ap1_fs references the same memory location as $ap1, and $ap2_fs references the same memory location as $ap2. Analogously, the second type of shadows are the *global-shadows* (or *g-shadows*), which are also introduced at the beginning of every function. For each global variable, there is one g-shadow per function, and each g-shadow is aliased with its corresponding global variable at the beginning of the function. For instance, if there are two global variables $x1 and $x2 in the program, then each function is assigned its own shadow variable $x1_gs for $x1, as well as a shadow variable $x2_gs for $x2. These definitions lead to the following properties of shadow variables:

- Shadow variables are local variables.
- Shadow variables cannot be accessed by the programmer, since they are fresh variables introduced by the analysis. This implies that they are never re-referenced after their initialization performed by the analysis.

Intuitively, the f-shadows of a function have the purpose of representing local variables of the caller that were aliased with a formal parameter of the function at the time of the call. Analogously, g-shadows represent local variables of the caller that were aliased with a global variable at the time of the function's invocation. This provides us with the means to determine how the aliases between the caller's local variables and global variables are modified by function calls.

```
01: a();
02: skip;   // u{(m.x1, m.x2)} a{}
03:
04: function a() {  // u{(m.x1, a.x1_gs)
05:                 //   (m.x2, a.x2_gs)} a{}
06:   $a1 =& $GLOBALS['x1'];
07:   skip;  // u{(m.x1, a.x1_gs, a.a1)
08:          //   (m.x2, a.x2_gs)} a{}
09:   b();
10:   skip;  // u{(m.x1, m.x2, a.x2_gs)
11:          //   (a.a1, a.x1_gs)} a{}
12: }
13:
14: function b() {   // u{(m.x1, b.x1_gs)
15:                 //   (m.x2, b.x2_gs)} a{}
16:
17:   $GLOBALS['x1'] =& $GLOBALS['x2'];
18:
19:   skip;   // u{(m.x2, b.x2_gs, m.x1)} a{}
20: }
```

Figure 9. Aliases between local variables and global variables.

To illustrate the value of shadow variables, consider Figure 9, which shows a code snippet covered by Case 1b. At the time of the call to function "b" on Line 9, the local variable $a1 is a must-alias of the global variable $x1. Inside the called function on Line 17, this global variable is re-referenced to another global variable. Without using g-shadows, the analysis would not be able to determine that $a1 is no longer aliased with $x1 when control flow returns to function "a" (remember that propagating local variables into the callee is not allowed). With the g-shadow, however, the analysis is able to extract this vital fact: In the information flowing back from function "b", the g-shadow of $x1 is not aliased with $x1 any more. Recalling the purpose of g-shadows, we know that the g-shadow of $x1 is indirectly representing $a1 (since $a1 was an alias of $x1 at the time of the call). Hence, we can deduce that $a1 is not aliased with $x1 any longer. Also, note that the fact that the global variable $x1 becomes an alias of the global variable $x2 is returned to the caller as well.

The detailed algorithm covering all presented cases can be found in Figure 15 in Appendix A. Due to space limitations, we abstain from a further discussion of the other cases. The interested reader is referred to our web site [17] for a comprehensive collection of examples that have been used to test our algorithms in practice. These examples clearly demonstrate the ability of our analysis to solve even difficult aliasing problems.

4.4.5 Limitations

Currently, the employed analyses provide no support for object-oriented features of PHP. This means that object or member variables never appear as elements of alias relationships. Besides, reference statements that contain arrays or array elements are not considered. However, this restriction did not appear to impart the results in our experiments. Also, note that this limitation only applies to alias analysis, whereas literal and taint analysis invest significant efforts into tracking the attributes of arrays and their elements. These limitations are the reason why our analysis is unsound (i.e., it may generate false negatives). For instance, a taint value that is propagated through alias relationships between array elements is not detected.

5. Resolving Includes

Virtually all web applications written in scripting languages such as PHP divide their code over several source files. These files are consolidated at run-time by means of file inclusion. A major difference compared to file inclusions in C and other languages is that the names of the included files need not be represented by static literals. Instead, these names can be composed of arbitrary expressions. Therefore, it is necessary to compute information about the value of these expressions to be able to take into account included files during static analysis. Straightforwardly applying a simple preprocessor such as the one used for C programs would not suffice, as it would leave a significant number of includes unresolved.

Basically, the task of resolving includes can be performed by literal analysis. A straightforward approach would be to include successfully resolved files "on the fly" during literal analysis. However, this results in the problem of having to modify the lattice of a running data flow analysis, which is both conceptually demanding and difficult to implement. Another issue is performance: It would be desirable to immediately resolve literal includes without the need to perform a fully-fledged literal analysis.

Our solution is to apply an iterative two-stage preprocessing step that is fast, precise, and easy to implement. In the first stage, we transitively resolve and include files whose names are directly given by literals (strings). In the second stage, if there are any non-literal include statements, we perform a literal analysis on the code that resulted from the first stage. This second stage may lead to the inclusion of additional files, which may again contain simple literal includes. Hence, we continue with the next iteration of the first stage and handle literal includes again. The process eventually terminates when there are no resolvable includes left.

PHP also permits the definition of recursive include relationships, which are used very rarely in practice. A simple approximative solution to this problem would be to include every file not more than once. Unfortunately, this would be highly imprecise because real-world applications often include the same files multiple times, even if there are no recursive includes. This practice is analogous to calling a function multiple times without calling it recursively. Therefore, during our include resolution process, we build an include graph that is used to determine whether an encountered include is recursive or not. Only in case of real recursive includes, we approximate such statements by treating them like no-ops.

6. Empirical Results

We used the presented concepts to enhance Pixy, the prototype system introduced in our previous paper [14], and performed a series of experiments to demonstrate its ability to detect previously unknown cross-site scripting vulnerabilities. To this end, Pixy was run on the current versions of three open source PHP programs. In contrast to C or Java programs, which have one clearly defined entry point where the execution starts (i.e., the main function), web applications written in PHP usually have several different entry points. These entry points correspond to the files visible in the browser's location bar while surfing the web application. We provided these entry points as input files to Pixy, which automatically resolved further file inclusions. Table 2 shows a summary of our results, including the number of entry points and the total lines of code that have been analyzed. To determine the line count, we do not factor out files that were analyzed multiple times in different contexts. For example, if an entry file "a.php" includes a file "b.php" twice, the lines of "b.php" are counted twice. Most entry files (together with their transitively included files) were analyzed in less than a minute using a 3.0 GHz Pentium 4 processor with 1GB RAM, even though our prototype still presents many opportunities for perfor-

mance tuning. There was no analysis run that took longer than five minutes.

In total, we discovered 106 exploitable XSS vulnerabilities in the latest versions of the analyzed programs. In all cases, we informed the authors about the issues and posted security advisories to the BugTraq mailing list [4]. The false positive rate of about 50 % is relatively low and further alleviated by the fact that many false positives are similar, which makes their recognition easier (see Section 6.2). Pixy also reported a few programming bugs not relevant for security, such as function calls with too many arguments. Since these bugs have no influence on program security, they were counted neither as vulnerabilities nor as false positives. These results clearly show that our analysis is capable of efficiently finding previously unknown vulnerabilities in real-world applications.

6.1 A Case Study: MyBloggie

Detailed descriptions of the discovered vulnerabilities are given in the corresponding BugTraq postings. In this section, we will take a closer look at an interesting vulnerability that we discovered in MyBloggie. This vulnerability is rather complex, especially when inspected in its original, unsimplified form. The relevant code spans three different source files and two functions, and includes value flows between parameters, arrays, and variables from different scopes. Finding such a vulnerability without the assistance of an automated analysis tool is quite unlikely.

Figure 10 shows the code in a simplified and condensed form. The sensitive sink on Line 11 receives a tainted value as input, which is held by the function's second formal parameter ($message). This function is called from Line 8 with $tbstatus as actual parameter. Inside the branches of the preceding if-construct, $tbstatus is either set to the empty string on Line 5 (which is untainted), or is built up from the variable $tbreply on Line 3 (the "." is PHP's string concatenation operator). The value of the global variable $tbreply is set by the call to function "multi_tb" on Line 2. A closer look at this function reveals that $tbreply is tainted whenever the first parameter $post_urls of function "multi_tb" is tainted. First, $post_urls is split into an array on Line 16. Afterwards, this array is traversed by the loop starting on Line 17. Inside the loop, $tbreply is assembled from the elements of the array $tb_urls. In effect, since $post_urls can be controlled directly by the attacker (through including the appropriate parameter in a request), this means that the described data flow chain eventually leads to control of the critical $message variable on Line 11.

As already mentioned in Section 4.3, the "global" keyword has the effect that a local variable is aliased with the corresponding global variable. Thus, without the help of alias analysis, we would not have been able to detect the value flow from $post_urls to $tbreply, leaving the described vulnerability undetected.

6.2 False Positives

A majority of the reported false positives (38 of 57) were due to impossible program paths. Figure 11 shows a simplified example of such a case, taken from DCP Portal. The analysis reported that the sensitive sink on Line 4 receives tainted input, namely the value returned by the call to function "SelectMember". The return value of this function may be equal to the global variable $site_name (Line 11). This global variable is initialized with an untainted value on Line 2 only if the condition on Line 1 evaluates to true. Closer inspection revealed that, in fact, this condition always evaluates to true in practice. Otherwise, it would mean that the underlying database would be seriously corrupted, which would hardly remain unnoticed by the administrators. This particular case was responsible for 13 false positives. As soon as we determined the reason for the first of these reports, it was easy to identify the remaining ones as false positives as well.

```
01: if (...) {
02:   multi_tb($post_urls, ...);
03:   $tbstatus = $tbstatus . $tbreply;
04: } else {
05:   $tbstatus ="";
06: }
07:
08: message(..., $tbstatus);
09:
10: function message(..., $message ) {
11:   echo $message;
12: }
13:
14: function multi_tb($post_urls, ...) {
15:   global $tbreply;
16:   $tb_urls = split('( )+', $post_urls, 10);
17:   foreach($tb_urls as $tb_url) {
18:     $tbreply .= $tb_url;
19:   }
20: }
```

Figure 10. Vulnerability in MyBloggie (simplified).

```
01: while ($row = mysql_fetch_array($result)) {
02:   $site_name = $row["site_name"];
03: }
04: echo SelectMember(..., ...)
05:
06: function SelectMember($id, $opt) {
07:   global $site_name;
08:   if (...) {
09:     ...
10:   } else {
11:    return $site_name;
12:   }
13: }
```

Figure 11. False positive due to impossible path (simplified).

As described by Sharir and Pnueli in [28], there are two types of context-sensitive interprocedural analyses, namely *call-string* analysis and *functional* analysis. Functional analysis usually provides more precise results than call-string analysis. In general, none of the two analyses is faster than the other *per se*, since their performance largely depends on the call graph of the analyzed program. However, for one entry point to MyBloggie, it turned out that functional analysis created a large number of contexts during the interprocedural analysis. To address this problem, we decided to perform taint analysis as instance of a call-string analysis with one-element call-strings for our experiments. The resulting twelve false positives could be eliminated by simply switching back to functional analysis. We believe that the imposed performance penalty of performing a functional analysis can be effectively reduced by refining some of the internal mechanisms of our analysis (such as the workset order, or the merging of equivalent contexts into one).

Five false positives were caused by variable array indices. For instance, the predefined PHP variables $_SERVER['PHP_SELF'] and $_SERVER['HTTP_HOST'] are untainted, since they cannot be controlled by an attacker. However, the value of some variable entries of $_SERVER (such as $_SERVER[$v]) are conservatively assumed to be tainted because there exist a few entries that can be controlled by an attacker. Using literal analysis to resolve such variable array indices could eliminate this type of false positive.

Program	Entry Files	LOC	Time (sec/File)	Vulnerabilities	False Positives	BugTraq ID
DCP Portal 6.1.1	22	61 617	6.0	61	35	427175
MyBloggie 2.1.3beta	6	20 326	58.3	14	5	427182
TxtForum 1.0.4-dev	15	4 398	1.3	31	17	427186, 427188
Totals	**43**	**86 341**	**11.7**	**106**	**57**	

Table 2. Summary of vulnerability reports.

```
include('lib/' . $_POST['fname'] . '.inc.php');
include($_POST['path'] . '/somefile.php');
```

Figure 12. A harmless and a dangerous unresolvable inclusion.

The remaining two false positives resulted from custom sanitization using regular expressions. Our prototype does not regard regex sanitization as reliable, since it is often difficult to cover all possible attack vectors (that is, it is easy to omit certain dangerous characters). Manual inspection, however, did not reveal any ways of circumventing the protection applied in these two cases.

6.3 File Inclusion Effectiveness

Table 3 summarizes our observations concerning the applied file inclusion algorithm. The second column lists the average number of iterations that were necessary for processing the entry files of a program (along with all their transitive inclusions). There was no entry file that required more than four iterations, and each entry was processed in less than 15 seconds. The third and fourth columns show the average number of literal and non-literal includes that were resolved per file. This demonstrates that non-literal includes occur more frequently than literal includes, and, as a result, the need for an intelligent resolution algorithm that is able to handle non-literal cases. Otherwise, a significant number of inclusions would be missed, leading to both false positives and false negatives.

All non-literal includes that could not be resolved assemble the names of the files to be included from dynamic input (mostly from user input, such as cookie fields and POST values, and sometimes from file contents). A close manual inspection of such cases is advisable, since they represent potential security leaks. If an attacker has control over the names of the files that are to be included, it might be possible to inject arbitrary scripts (i.e., arbitrary PHP code) into the program. Most of the cases we encountered are harmless and similar in structure to the first inclusion shown in Figure 12. In this example, it is impossible to include a remote file (e.g., located on the attacker's server) because the name of the included file starts with "lib", and not with a protocol specifier such as "http://". However, it would still permit path traversal attacks through the use of path strings containing elements such as "../..". For instance, an attacker could trick the statement into including the server's "/etc/passwd" file, which would be returned verbatim by PHP. This threat is mitigated by the provided suffix ".inc.php", resulting in the restriction that only files with this extension are included. In one case, however, an include statement such as the second one shown in Figure 12 was encountered. Here, an attacker can cause the inclusion of an arbitrary remote script with the name "somefile.php". By placing such a file on a web server under the attacker's control and providing this file's URL in the POST parameter "path", the code contained inside this file (written by the attacker) is executed with the privileges of the running PHP server.

7. Related Work

Currently, there exist only few approaches that deal with static detection of web application vulnerabilities. Huang et al. [13] were the first to address this issue in the context of PHP applications. They used a lattice-based analysis algorithm derived from type systems and typestate, and compared it to a technique based on bounded model checking in their follow-up paper [12]. Alias analysis or include file resolution is not performed.

A recent technical report by Xie and Aiken [33] addresses the problem of statically detecting SQL injection vulnerabilities in PHP scripts. By applying a custom, three-tier architecture instead of using fully-fledged data flow analysis techniques, they operate on a less ambitious conceptual level than we do. For instance, recursive function calls are simply ignored, and no alias analysis is performed. The authors briefly mention an approach to resolve include statements that seems to yield good results in practice. Unfortunately, comparing their approach to ours is difficult due to the lack of a more detailed description. For instance, the problem of recursive or non-literal includes is not addressed explicitly.

Livshits and Lam [20] applied an analysis supported by binary decision diagrams developed by Whaley and Lam [31] for finding security vulnerabilities in Java applications. Their work differs from ours in the underlying analysis, which is flow-insensitive for the most part, and the target language, which is typed. This considerably eases the challenges faced by static analysis.

Minamide [21] presented a technique for approximating the string output of PHP programs with a context-free grammar. While primarily targeted at the validation of HTML output, the author claims that it can also be used for the detection of XSS vulnerabilities. However, without any taint information or additional checks, it appears to be difficult to distinguish between malicious and benign output. Only one discovered XSS flaw is reported, and the observed false positive rate is not mentioned. Moreover, only "basic features" of PHP are supported, excluding references.

Engler et al. have published various static analysis approaches to finding vulnerabilities and programming bugs in C programs. In [7], the authors describe a system that translates simple rules into automata-based compiler extensions that check whether a program adheres to these rules or not. In an extension to this work, the authors present techniques for the automatic extraction of such rules from a given program [8]. Finally, tainting analysis is used to identify vulnerabilities in operating system code where user supplied integer and pointer values are used without proper checking [3].

An alternative approach aiming at the detection of taint-style vulnerabilities introduces special type qualifiers to the analyzed programming language. One of the most prominent tools that applies this concept is CQual [9], which has been, among other things, used by Shankar et al. [27] to detect format string vulnerabilities in C code. However, it remains an open question whether this technique can be applied to untyped scripting languages.

8. Conclusion

Web applications have become a popular and wide-spread interaction medium in our daily lives. At the same time, vulnerabilities that endanger the personal data of users are discovered regularly. Manual security audits targeted at these vulnerabilities are labor-intensive, costly, and error-prone. In a previous paper [14], we proposed a precise static analysis technique that is able to detect the

Program	Iterations	Resolved Literal Includes	Resolved Non-Literal Includes	Unresolved Includes
DCP Portal 6.1.1	3.9	0.9	5.8	1.9
MyBloggie 2.1.3beta	3	6.5	12.3	0
TxtForum 1.0.4-dev	1.5	1.8	2.6	1.7

Table 3. Summary of file inclusions (average numbers).

broad class of taint-style vulnerabilities automatically. Our analysis was based on data flow analysis, a well-understood and established technique in computer science. It tackled several issues specific to scripting languages such as PHP, which make these languages harder to analyze statically.

In this paper, we enhanced our previous work by integrating a novel alias analysis using shadow variables. In contrast to the alias analysis we applied previously, our new approach is able to generate precise solutions even for difficult aliasing problems. Moreover, we presented an iterative, two-stage preprocessing step for the automatic resolution of file inclusions. We tested our concepts by running our improved analysis tool on three web applications. The empirical results show that we are able to efficiently and automatically detect vulnerabilities with a low false positive rate.

There is an urgent need for automated vulnerability detection in web application development, especially because web applications are growing into large and complex systems. We believe that our presented techniques improve state-of-the-art solutions to this problem, offering benefits to both users and providers of web applications.

9. Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grant P18368-N04.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

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

[3] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, 2002.

[4] BugTraq. BugTraq Mailing List Archive. `http://www.securityfocus.com/archive/1`, 2005.

[5] CERT. CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2005.

[6] David Chase, Mark Wegman, and F. Ken Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1991.

[7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 2000*, 2000.

[8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[9] Jeffrey S. Foster, Manuel Faehndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.

[10] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[11] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, 2003.

[12] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *DSN*, 2004.

[13] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th International Conference on World Wide Web*, 2004.

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[15] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report). `http://www.seclab.tuwien.ac.at/projects/pixy/`, 2006.

[16] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *The 21st ACM Symposium on Applied Computing (SAC 2006)*, 2006.

[17] Secure Systems Lab. Secure Systems Lab, Technical University of Vienna. `http://www.seclab.tuwien.ac.at`, 2006.

[18] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 1992.

[19] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2005.

[20] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, August 2005.

[21] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, 2005.

[22] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP Security 2005*, 2005.

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

[25] PHP. PHP: Hypertext Preprocessor. `http://www.php.net`, 2005.

[26] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.

[27] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[28] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7. Prentice-Hall, 1981.

[29] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[30] Larry Wall, Tom Christiansen, Randal L. Schwartz, and Stephan Potter. *Programming Perl (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[31] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[32] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995.

[33] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. `http://glide.stanford.edu/yichen/research/sec.ps`, 2006.

A. Algorithms

```
function combine (AliasInfo input-1, AliasInfo input-2) {
 - AliasInfo output;
 - output.may-aliases =
   union of may-alias pairs of input-1 and input-2
 - for each must-alias-group in input-1:
   - in an auxiliary graph, create a strongly connected component
     consisting of the group members
 - for each must-alias-group in input-2:
   - in the auxiliary graph, create a strongly connected
     component consisting of the group members; if an edge to
     be drawn already exists, promote it to a double edge
 - for each normal (i.e., single) edge in the graph:
   - add the may-alias-pair containing the corresponding nodes
     to the output information
 - for each strongly connected component that contains
   only double edges:
   - add the must-alias group containing the corresponding
     nodes to the output information
 - return output information
}
```

Figure 13. Algorithm for the combination operator.

```
- for each call-by-reference pair:
  - create a placeholder variable for the formal parameter
    and add it to the actual parameter's must-alias group
  - for each may-alias pair that contains the actual parameter:
    - copy this pair, replace the actual parameter in the new pair
      by the formal parameter's placeholder, and add the new
      pair to the set of may-aliases
  - remove all local variables that belong to the caller
  - remove all must-alias groups and may-alias pairs that
    have only one element
  - replace the placeholders by the corresponding
    formal parameters
}
```

Figure 14. Algorithm for adjusting the alias information that is propagated into a callee.

```
- origInfo: the information entering the call node
- localInfo: contains only the aliasing information between
  locals of the caller (extracted from origInfo)
- interInfo: contains only the aliasing information between globals
  (taken from the information at the end of the callee)
- outputInfo: initialized with localInfo and interInfo;
  the following steps compute and add the aliases between
  global variables and local variables;
  results in the information at the local exit of the call node

// G-Shadows: Must-Aliases
- for each must-alias group in origInfo:
  - if it contains at least one local variable v
    and at least one global variable g:
    - mark this group as visited
    - if the g-shadow of g has at least one
      global must-alias g_u at the end of the called function:
      - in outputInfo, merge the must-alias group containing v
        with the must-alias group containing g_u (also
        considering implicit one-element groups)
    - for each global may-alias g_a of the g-shadow at
      the end of the called function:
      - add the may-alias-pair (v, g_a) and all
        may-alias-pairs (v_u, g_a) to outputInfo,
        where v_u denotes "each local must-alias of v"

// G-Shadows: May-Aliases
- for each may-alias pair containing a local and a global
  in origInfo:
  - for each global alias (both must and may)
    of the global's g-shadow at the end of the callee:
    - add the may-alias pair (local, alias) to outputInfo

// F-Shadows: Must-Aliases and May-Aliases
- for each local actual call-by-reference parameter p:
  - determine the corresponding formal's f-shadow fs
  - find p's must-alias group in origInfo
    (also considering implicit one-element groups)
  - if this group is not marked as visited:
    - mark the group as visited
    - if the f-shadow fs has at least one global must-alias f_u
      at the end of the callee:
      - in outputInfo, merge the must-alias group containing
        p with the must-alias group containing f_u
        (also considering implicit one-element groups)
    - for each global may-alias f_a of the f-shadow at
      the end of the callee:
      - add the may-alias pair (p, f_a) and the may-alias-pairs
        (p_u, f_a) to outputInfo , where p_u denotes
        "each local must-alias of p"
  - for each local may-alias lma of p:
    - for each global alias (both must and may) of
      the f-shadow at the end of the called function:
      - add the may-alias pair (lma, alias) to outputInfo
```

Figure 15. Algorithm for computing the alias information after a function call.