Journal of Computer Security 18 (2010) 861–907
DOI 10.3233/JCS-2009-0385
IOS Press

Static analysis for detecting taint-style vulnerabilities in web applications

Nenad Jovanovic [a], Christopher Kruegel [b,*] and Engin Kirda [c]

[a] *Secure Systems Lab, Technical University Vienna, Vienna, Austria*
E-mail: enji@seclab.tuwien.ac.at
[b] *UC Santa Barbara, CA, USA*
E-mail: chris@cs.ucsb.edu
[c] *Institut Eurecom, Sophia Antipolis, France*
E-mail: kirda@eurecom.fr

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. More precisely, we use flow-sensitive, interprocedural and context-sensitive data flow analysis to discover vulnerable points in a program. In addition to the *taint analysis* at the core of our engine, we employ a 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 an iterative two-phase algorithm for fast and precise resolution of file inclusions. The presented concepts are targeted at the general class of taint-style vulnerabilities and can be easily applied to the detection of vulnerability types such as SQL injection, cross-site scripting (XSS), and command injection.

We implemented the presented concepts in Pixy, a high-precision static analysis tool aimed at detecting cross-site scripting and SQL injection vulnerabilities in PHP programs. To demonstrate the effectiveness of our techniques, we analyzed a number of popular, open-source web applications and discovered hundreds of previously unknown 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.

Keywords: Program analysis, static analysis, data flow analysis, alias analysis, web application security, scripting languages security, cross-site scripting, SQL injection, 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 heightened importance of web applications, the negative impact of security flaws

*Corresponding author: C. Kruegel, Department of Computer Science, University of California, Santa Barbara, CA 93106, USA. Tel.: +1 805 893 6198; Fax: +1 805 893 8553; E-mail: chris@cs.ucsb.edu.

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 [23], 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., [15,29,33], and Perl's taint mode [41]) try to detect attacks while executing the audited program, whereas static analyzers [16,17,20,26,27,42,46] scan the entire web application's source code for vulnerabilities before it is deployed.

In this paper, we present Pixy, a tool for statically detecting XSS and SQL injection vulnerabilities in PHP 4 [32] code by means of data flow analysis. We chose PHP as target language since it is widely used for creating web applications [37], and a substantial number of security advisories refer to PHP programs [4]. Of course, the general concepts can be equally applied to other *taint-style* vulnerabilities, for example, command injection attacks (see Section 2). The main contributions of this paper are as follows:

- A flow-sensitive, interprocedural, and context-sensitive data flow analysis for PHP, targeted at detecting taint-style vulnerabilities. This analysis process had to overcome significant conceptual challenges due to the untyped nature of PHP.
- A precise alias analysis targeted at the unique reference semantics commonly found in scripting languages. This analysis generates precise results even for conceptually difficult aliasing problems. Without a preceding alias analysis, taint analysis would generate false positives as well as false negatives in conjunction with aliases.
- We enhance the quality and quantity of the generated vulnerability reports as well as our tool's usability by integrating an 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, which requires more sophisticated resolution techniques.
- We present empirical results that demonstrate that our tool can be used to detect XSS and SQL injection vulnerabilities in real-world programs. The analysis process is fast, completely automatic, and produces few false positives.

2. Taint-style vulnerabilities, XSS attacks and SQL injection

The presented work 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 taint-style vulnerabilities is given by Livshits and Lam in [26]. Section 12 discusses related work that attempts to detect this type of vulnerability.

2.1. Cross-site scripting (XSS)

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 a 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 *same-origin policy*. That is, a JavaScript program has access only to cookies that belong to the site from which the code originated.

XSS attacks circumvent the same-origin policy 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 following simple PHP script, where a user's search query is displayed after submitting it:

```
echo "You searched for " . $_GET['s'];
```

The user's search query 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?s=<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. As soon as the user clicks on this link, her browser visits the page `post.php` on the vulnerable site, with the GET parameter "s" set

to the malicious JavaScript code. As a result, the malicious code is embedded in the application's reply page, and now has access to the user's cookie. The JavaScript code sends the cookie to the attacker, who can now use it to impersonate the victim.

The particular type of XSS vulnerability discussed above is called *reflected* XSS, since the attacker's malicious input is immediately returned (i.e., reflected) to the victim. There also exists a second type of XSS, where the application first stores the input into a database or the file system. At a later stage, the application retrieves this data through database queries or file reads, and finally sends it to the victim. For instance, such *stored* XSS vulnerabilities often occur in web guestbooks or forums, where a visitor leaves a comment that is later accessed by another visitor.

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*: PHP functions such as htmlentities() and 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().

Currently, our tool can only handle reflected XSS vulnerabilities. However, it is straightforward to use it for the detection of stored XSS as well, given a certain program policy with regard to the taint status of persistently stored data. For instance, it is customary that data is not sanitized before it is stored to a database or to the file system, which means that it has to be sanitized after its later retrieval. In our system, this can be modeled by adding the corresponding data retrieval functions to the set of entry points. Analogously, the application's policy can demand that all data is sanitized *before* it is stored. In this case, data storage functions have to be defined as sensitive sinks. Mixed policies are more difficult to handle. For instance, an application could expect a certain database table to contain only sanitized values, whereas some other table might also be allowed to contain unsanitized values. Here, the analysis would also have to resolve the names of the tables that are used for storage and retrieval. To the best of our knowledge, there exist no studies that answer the question which of these policies is prevalent in real-world programs.

2.2. SQL injection (SQLI)

Many web applications make use of a backend database for storing data such as client accounts, postings, or user preferences. The interaction with the database is typically performed by means of SQL queries. These queries are often assembled dynamically by the program, which combines predefined SQL language elements with user input. For instance, the query in the following PHP snippet attempts to

retrieve an account based on a name and password that are provided by the user through an HTTP GET request:

```
mysql_query("SELECT * FROM users WHERE name='$_GET[name]'
AND pw='$_GET[pw]'");
```

An SQL injection vulnerability occurs whenever an attacker is able to alter the syntactic structure of such queries in an unexpected way [40]. Given the example above, by providing an empty `name` parameter and a specially crafted `pw` parameter (underlined), an attacker could trick the application into issuing the following query to the database:

```
SELECT * FROM users WHERE name='' AND pw='' OR name='admin
```

The effect of the altered query is that the account of the user "admin" is returned, bypassing the application's authentication mechanism. In other cases, SQLI attacks can also seriously corrupt the backend database, or even compromise the entire host that the web application runs on.

SQL injection can be roughly described by the following properties:

- *Entry points* into the program: GET, POST and COOKIE arrays.
- *Sanitization routines*: PHP functions, such as mysql_escape_string(), that escape parts of the input that could interfere with the intended meaning of a SQL query.
- *Sensitive sinks*: All routines that use data to perform SQL queries, for example, mysql_query().

3. Data flow analysis

The goal of our analysis is to determine whether it is possible that tainted data reaches sensitive sinks without being properly sanitized. To this end, we have to identify the taint value[1] of variables that are used in these sinks. For this, we apply the technique of data flow analysis, which is a well-understood topic in computer science and has been used in compiler optimizations for decades [1,28,30]. 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, *literal analysis* computes, for each program point, the literal values that variables may hold.

Note that we use the term "literal analysis" to refer to the classic combination of constant propagation and constant folding. The reason is that in PHP terminology,

[1] The taint value of a variable determines whether the variable is tainted or not.

the word "constant" has a meaning that is different from the usual meaning (where it denotes a literal value). In PHP, a constant is a special type of variable that can only hold simple values (e.g., it cannot contain array elements), and that cannot change after it has been defined. To avoid confusion due to the collision of the two different meanings of the word "constant", we will use the term "literal analysis" in this paper.

To illustrate how data flow analysis is used to perform literal analysis, imagine a fictitious programming language that uses only one variable (v) and two literals (the integers 3 and 4). Data flow analysis operates on the control flow graph (CFG) of a program. Figure 1 shows the CFG for a simple example program. In this figure, each CFG node is associated with its final data flow information after the analysis has finished. "Skip" nodes represent empty instructions. Assume further that the condition of the "if" branch cannot be resolved statically, for example, because it depends on the value of an environment variable. We use the symbol $\top$ ("top") for the *unknown literal*, which indicates that the exact value of the literal cannot be determined. This is the case on program entry, since in our programming language, a variable contains random garbage before being initialized. After performing literal analysis for this program, each CFG node is associated with information about which literal is mapped to variable v *before* executing that node. Note that the exact value for variable v after the "if" construct is also unknown, because the analysis cannot determine which branch will be taken at runtime. Inside the branches, however, precise information is available.

An important concept used in the theory of data flow analysis is that of a *lattice*,[2]

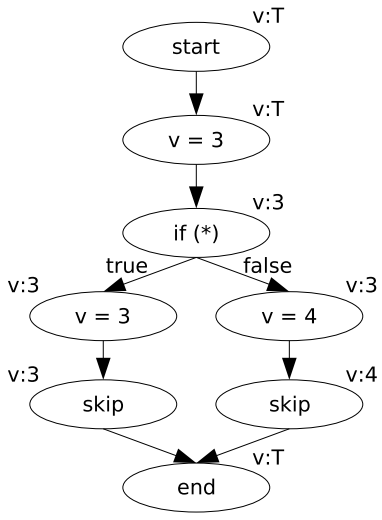

Fig. 1. Example CFG with associated resulting analysis information.

[2]To be precise, data flow analysis requires a complete lattice that satisfies the ascending chain condition, or equivalently, a semi-lattice with a unit that satisfies the ascending chain condition [30].

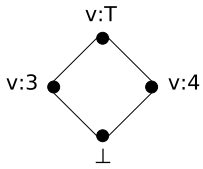

Fig. 2. A simple lattice.

which is used to represent the type of information that is to be collected. Every piece of information that could ever be associated with a CFG node by the analysis must be contained as an element of the used lattice. The lattice for our previous example is depicted in Fig. 2. Note that an additional *bottom element* $\bot$ is required by the analysis algorithm for marking nodes as "not visited yet" at the beginning. Each line in Fig. 2 indicates an ordering between the elements at its end points with regard to precision. For instance, the element $(\mathrm{v}{:}3)$ is more precise (less conservative) than $(\mathrm{v}{:}\top)$, written as $(\mathrm{v}{:}3) \sqsubset (\mathrm{v}{:}\top)$. In this sense, the bottom element is defined as being smaller than all other elements. The *least upper bound* $\sqcup$ of two elements is the smallest element that is greater than or equal to both of the elements. For example, $(\mathrm{v}{:}3) \sqcup (\mathrm{v}{:}4) = (\mathrm{v}{:}\top)$, and $(\mathrm{v}{:}3) \sqcup (\bot) = (\mathrm{v}{:}3)$. The $\sqcup$ operator is used for conservatively combining the information of merging paths (e.g., after "if" branches), which can also be seen at the end node in Fig. 1.

In this paper, the notation with regard to lattice structure and the least upper bound operation follows Nielson et al. [30]. Other authors, especially in the advent of data flow analysis (introduced by Kildall [22]), used an alternative, but equivalent notation. In this other notation, more precise elements were located at the top of the lattice, instead of at its bottom.

Another important ingredient for data flow analyses are *transfer functions*. Each CFG node is associated with such a transfer function, which takes a lattice element as input and returns a lattice element as output. The purpose of transfer functions is to model the semantics of their corresponding node with respect to the collected information. In our example in Fig. 1, all assignment nodes possess a transfer function that adjusts the literal mapping of the assigned variable accordingly.

After specifying a data flow analysis with its underlying lattice and transfer functions, an iterative algorithm for computing the results is initiated. Beginning at the program's entry node, this algorithm propagates analysis information through the program by applying transfer functions and combining information at merge points. As soon as a fixed point is reached, where additional computations do not lead to changes anymore, the algorithm terminates, and the analysis is finished.

A *flow-sensitive* analysis considers the ordering of program instructions, whereas a *flow-insensitive* analysis does not (i.e., reordering instructions inside a function does not affect its results). *Interprocedural* analyses take function calls into account, while *intraprocedural* analyses operate only inside a single function. *Context-sensitive* in-

terprocedural analyses are able to distinguish between different call sites to a function, *context-insensitive* interprocedural analyses are not, and, therefore, confuse the information computed for different calls to the same function. Hence, a high precision can be achieved by performing an analysis that is flow-sensitive, interprocedural, and context-sensitive. Our presented scanner is equipped with a combination of all these desirable features. Of course, precision can always be further enhanced (with the trade-off that performance is reduced). For instance, it would be possible to define an analysis that is *path-sensitive*, such that impossible paths due to contradicting guard conditions of program branches can be recognized.

An analysis is *sound* if the information it generates is a safe approximation of the real information. In the context of vulnerability scans, a sound analysis must include all vulnerabilities in its report. *Completeness* denotes the opposite notion, meaning that a complete analysis must not generate any false alarms (*false positives*). Our system is unsound with respect to the language features mentioned in Section 9.4 (most notably, object-orientation), because these features are modeled in an optimistic way. That is, we apply a simple approximation of these language constructs, assuming that no tainted values can originate from there. At the same time, our system is incomplete, as it may generate false warnings due to the conservative nature of data flow analysis. A detailed discussion of the generated false positives will be presented in Section 11.

Sometimes, the terms "sound" and "complete" are used in a reversed way, depending on how the underlying mathematical notions are translated into the context of vulnerability analysis. In this paper, we follow the interpretation used in [19] and [36], which corresponds to the explanations given above.

Note that the theory of *abstract interpretation* is closely related to data flow analysis, although it is formulated on a higher level. A comprehensive overview of abstract interpretation, data flow analysis and other program analyses is given in [30], and a vivid introduction to the topic can be found at [31].

4. PHP front-end

In order to conduct static analysis, the input program has to be parsed and transformed into a form that makes the following analysis as easy as possible. This first step includes the linearization of arbitrarily deep expressions in the original language as well as the reduction of various loop and branch constructs such as "foreach" and "switch" to combinations of "if"s and "goto"s. The resulting intermediate representation, *P-Tac*, resembles the classic *three-address code* (TAC) presented in [1]. TAC is an assembly-like language characterized by statements with at most three operands and the general form "x = y op z". For example, the input statement "a = 1 + b + c" would be translated into the corresponding TAC sequence "t1 = 1 + b; t2 = t1 + c; a = t2". The variables t1 and t2 are temporaries introduced in the course of the translation, and do not appear elsewhere in the program.

4.1. Parse tree construction

As the first step towards the desired intermediate representation, the input PHP code is parsed and stored as a parse tree for further processing. PhpParser, our tool for generating parse trees for PHP code, is a combination of the Java lexical analyzer JFlex [18], the Java parser Cup [8], and the Flex and Bison specification files from the sources of the PHP interpreter [32]. Due to a few incompatibilities of Flex and Bison with their Java counterparts, we carefully modified JFlex and Cup in order to accept the original specification files and to permit the convenient definition of Java actions for constructing the parse tree. The PhpParser package contains a more detailed documentation of these modifications.

4.2. Intermediate representation: P-Tac

As mentioned previously, the constructed parse tree is transformed into *P-Tac*, a linearized form of the original PHP script resembling three-address code [1], and kept as a control flow graph for each encountered function. The code in the global scope (external to all user-defined functions) is moved into a special "main" function, which represents the starting point of the PHP script. All loop constructs (while, for, switch) are replaced by equivalent "if"-branches in order to prevent unnecessary redundancies in the following analysis. *Constants* in PHP are specified with the built-in "define" function, whereas values such as 77 or "foo" will be termed as *literals*. Table 1 gives an overview of the most important CFG nodes created by the P-Tac converter. The *place* abstraction denotes variables, constants, and literals and was introduced to permit more concise specifications. Function calls are represented by three CFG nodes to make the design of interprocedural transfer functions easier (a call preparation node, the actual call node, and a call return node). Calls to functions for which no definition was found are replaced by calls to the special *unknown*

Table 1
Main types of CFG nodes in P-Tac

CFG Node	Shape/Description
Simple assignment	{var} = {place}
Unary assignment	{var} = {op} {place}
Binary assignment	{var} = {place} {op} {place}
Array assignment	{var} = array()
Reference assignment	{var} &= {var}
Unset	unset({var})
Global	global {var}
Call preparation	A call node's predecessor
Call	Represents a function call
Call return	A call node's successor

function. During taint analysis, calls to this function are approximated in a conservative way, meaning that the return value is considered to be tainted. Analogously, literal analysis considers the return value to be unknown ($\top$). Alias analysis treats such calls as no-op statements, which have no effect on the computation of alias information.

5. Analysis back-end

A straightforward approach to solving the problem of detecting taint-style vulnerabilities would be to immediately conduct a *taint analysis* on the intermediate representation generated by the front-end. This taint analysis would identify points where tainted data can enter the program, propagate taint values along assignments and similar constructs, and inform the user of every sensitive sink that receives tainted input. However, to enable the analysis to produce correct and precise results, significant preparatory work is required. For instance, whenever a variable is assigned a tainted value, this taint value must not be propagated only to the variable itself, but also to all its aliases (variables pointing to the same memory location). Hence, information about alias relationships has to be provided by a preceding *alias analysis*. Moreover, the dynamic nature of PHP's file inclusion mechanism requires a *literal analysis* for resolving the names of the files that are to be included (see Section 10). These three components (literal analysis, alias analysis, and taint analysis) will be discussed in the following sections separately.

One of the key features of our analysis is its high precision, since it is flow-sensitive, interprocedural, and context-sensitive. Moreover, we are the first to give a detailed description of how to perform an alias analysis for an untyped, reference-based scripting language such as PHP. Although there exists a rich literature on C pointer analysis, it is questionable whether these techniques can be directly applied to the semantically different problem of alias analysis for PHP references. As mentioned by Xie and Aiken in [46], static analysis of scripting languages is regarded as a difficult problem and has not achieved much attention so far. In this context, even apparently trivial issues such as the simulation of the effects of a simple assignment require careful considerations. For instance, multi-dimensional arrays can contain elements that are neither explicitly addressed nor declared. To correctly handle the assignment of such a multi-dimensional array to another array variable, these hidden elements must be taken into account. The following sections will address these issues in detail.

6. Literal analysis: Basics

The purpose of literal analysis is to determine, for each program point, the literal that a variable or a constant can hold. This information can be used for improving

the precision of the overall analysis in various ways. For instance, our prototype uses the computed information for resolving the names of files to be included. Other potential uses of literals information are the resolution of variable variables, variable array indices, and variable function calls.

6.1. Carrier lattice definition

As mentioned in Section 3, an important building block for data flow analyses is the underlying lattice. The lattice used for literal analysis basically resembles the simple lattice for our toy programming language that was introduced in Fig. 2, with two differences. First, the literal analysis lattice does not provide mappings for a single variable, but for all variables and constants that appear in the scanned program. Second, it is able to describe the mapping to any possible literal, and not just to the literals 3 and 4. Since the number of possible literals is infinite, this means that the "breadth" of the lattice (when shown as Hasse diagram [44]) is infinite as well. Figure 3 shows a fragment of a lattice for a program with two variables ($a and $b) and one constant (CONST) to provide an intuitive feeling for the ordering among lattice elements. Note that the lower two elements differ only in the value that the variable $a is holding. Hence, the least upper bound of these two elements is identical except that it maps $a to $\top$. The top element of the whole lattice (which is not depicted in Fig. 3) maps all variables and constants to the unknown literal $\top$, meaning that we know "absolutely nothing". As in the previous simple lattice, the bottom element (not depicted either) is just a special placeholder element needed by the analysis algorithm.

6.2. Transfer functions definition

After defining the underlying lattice for the analysis, each CFG node has to be associated with a transfer function. These transfer functions determine how the analyzed information is affected when control flows through the corresponding CFG node. That is, it takes the lattice element entering the node as input, and returns a lattice element reflecting the node's semantics as output. The most straightforward example for literal analysis is a node of the form "simple_variable = literal", with the term *simple variable* denoting a variable that is neither an array

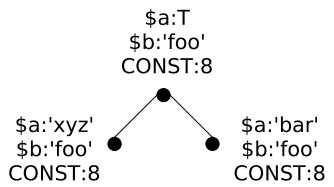

Fig. 3. Fragment of a literal analysis lattice.

nor an array element. For such a node, the transfer function only has to adjust the literal mapping of that variable. For example, the statement "$a = 5" assigns the literal value 5 to the variable $a in the lattice. For CFG nodes such as "simple_variable = variable", the assigned literal is not immediately available, but has to be extracted from the incoming lattice element by inspecting the mapping of the variable on the right side. Nodes like "simple_variable = constant" can be treated analogously to "simple_variable = variable".

Additional complexity arises when taking arrays, array elements, and non-literal array indices into account. The reason is that PHP is an untyped language without explicit type declarations. That is, it provides no explicit information about whether a variable is an array or not. A possible solution to this problem would be to perform an additional type inference. In our system, we take an alternative approach, which has turned out to be sufficient in practice. It is based on a simple, syntax-based detection of arrays and array elements. More precisely, our analysis considers a variable to be an array if it is indexed at some point in the program. For instance, if the expression "$a[2]" appears somewhere in the program, then variable "$a" certainly is an array. At the same time, the corresponding index to the array ($a[2]) obviously is an array element. Unfortunately, the absence of such expressions does not necessarily imply that a variable is not an array. This is demonstrated in Fig. 4. Here, $b is never indexed, but is still an array due to the assignment of array $a. This means that the intuitive introductory examples involving "simple_variable" need to be extended to deal with this issue, as we have no guarantee that a variable is really simple. Array elements do not suffer from this uncertainty: While $a[1] surely is an array element, $a is not.

In the course of our work, we found that the problem space regarding arrays and array elements can be divided into four cases. These cases depend only on characteristics of the variable on the *left* side of the assignment. An overview of these cases is given in Table 2, which also contains information about taint analysis and aliases. These topics will be dealt with later in this paper and can be ignored for the moment. Note that the four cases cover *all* possible situations that can occur in a program (modulo the limitations with regard to object-orientation discussed in Section 9.4). In the rest of this section, we will discuss the four cases in order of increasing complexity.

The simplest, first case applies when the left variable is not an array element (which can be easily decided because there is no array subscript) and not known

```
1 $a[1] = 7;  // $a obviously is an array
2 $b = $a;    // $b is a hidden array:
3             // it is not indexed anywhere
4 $c = $b;
5 echo $c[1]; // $c obviously is an array
```

Fig. 4. Arrays can be hidden.

Table 2
Actions performed by literal analysis and taint analysis for simple assignment nodes depending on the left-hand variable

Left variable	Literal analysis	Taint analysis
Not an array element and not known as array ("normal variable")	strong update for must-aliases, weak update for may-aliases	strong update (taint, CA flag) for must-aliases, weak update (taint, CA flag) for may-aliases
Array, but not an array element	strong overlap	target.caFlag = source.caFlag; strong overlap (taint)
Array element (and maybe an array) without non-literal indices	strong overlap	target.root.caFlag $\sqcup=$ source.caFlag; strong overlap (taint)
Array element (and maybe an array) with non-literal indices	weak overlap for all MI variables	target.root.caFlag $\sqcup=$ source.caFlag; weak overlap (taint) for all MI variables

as array.[3] Here, the analysis proceeds just as in the introductory "simple_variable" example, without taking into account whether the variable on the right side might have array elements or not. This punctual overwrite operation is called *strong update*. Note that when using this simple approach, precision might suffer. For instance, in Fig. 4, the analysis cannot determine that $c[1] is mapped to the literal 7 at the end of the program. However, this issue did not lead to significant losses in precision during our experiments.

The second of the four cases that need to be distinguished is when the left variable is an array, but not an array element. A useful concept in this respect is that of an *array tree*, which describes an array and its contents as a tree. The array variable itself is the tree's root, the array's elements are interior nodes and leaves, and its indices are edge labels. For example, Fig. 5 shows the tree for a two-dimensional array with the elements $a[$i][2] and $a[3][4]. Literals can be associated with each node to represent the current mappings. Intuitively, the analysis has to "overlap" the array tree of the left variable with the array tree of the right variable such that literals for matching nodes are overwritten. For instance, in the course of the assignment of array $b to array $a ($a = $b), the literal of $b[1] must overwrite the literal of $a[1]. The literals of nodes on the left side for which there is no matching node on the right side generally have to be set to $\top$ because it is uncertain whether there *really* is no matching node (due to the possibility that the array might contain hidden elements). In Fig. 4, for example, $c[1] is set to $\top$ after the assignment in Line 4 because the analysis cannot determine whether there is an element $b[1] or not. An exception to this rule can be made if there is a literal or a constant on the right side of the assignment. Since literals and constants can never be arrays, the analysis is free to set the literals of all array elements on the left side to NULL (which corresponds to the

[3] Note that we use the phrase "not known as array" to include the possibility that a variable might be a "hidden" array.

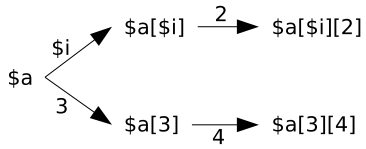

Fig. 5. Array tree example.

```
strongOverlap(Variable target, Place source) {
  if source known as array
    if target known as array
      for all direct elements of target
        if there is a direct element of source
        with the same direct index
          strongOverlap(target element,
                        source element)
        else
          set the array tree of the
          direct element to ⊤
  else if source is literal or constant
    set the array tree below target to NULL
  else
    set the array tree below target to ⊤

  set the target literal to the source literal
}
```

Fig. 6. Strong overlap algorithm.

actual semantics in PHP). A recursive algorithm for performing all these operations is given in Fig. 6, where the term *direct element* denotes an array element that is retrieved by adding a single index (the *direct index*) to its enclosing array.

Cases three and four require the understanding that array elements can have one or more non-literal indices. An example would be $a[1][$i][2], which has one non-literal and two literal indices. Such *non-literal array elements* have to be treated in a special way because they can represent multiple variables (such as $a[1][8][2] or $a[1][9][2] for the given example). Our analysis handles them in a pessimistic way in that they are mapped to $\top$. Otherwise, it would be necessary to track the non-literal indices of these elements and recompute the taint value of a non-literal array element whenever one of its indices changes. For example, if the variable $i changes at some point in the program, the analysis would have to recompute the values for all array elements with at least one index containing $i (such as $a[$i] or $b[$a[$i]]). Our experiments did not suggest that this additional complexity would lead to a significant gain in precision.

In the third case, the left variable is an array element without non-literal indices. This variable may also be an array (i.e., it does not matter whether it is known as array or not). This case is handled in the same way as case two, using the strong overlap algorithm. Note, however, that taint analysis (discussed in Section 9) will perform different operations for these two situations.

In the fourth and last case, the left variable is an array element with non-literal indices and maybe an array. As already mentioned, this case has to be treated separately because it is not certain which array element is actually meant by the non-literal array element. For instance, when assigning the literal 3 to $a[$i], the analysis has to consider that this might affect $a[8], $a[9], or any other element. So, instead of overwriting the literals of the possibly affected variables with the literal 3, we have to conservatively replace them with the least upper bound of the old and the new literal. For example, if the old literal of $a[8] was 7, it becomes $\top$. If it was 3, it remains 3. This approach can be formulated in a succinct way using the terms *MI variables* and *weak overlap*. The *MI variables* of a non-literal array element are all variables that are *maybe identical* to this array element. For example, $a[8] and $a[9] are MI variables of $a[$i]. These are the variables that might be affected by an assignment, and hence, have to be conservatively updated by the analysis. The *weak overlap* algorithm is analogous to the strong overlap algorithm, with the difference that all overwrite operations are replaced by least upper bound operations. This way, the analysis can handle assignments of the fourth type by performing a weak overlap for all MI variables.

6.3. Dependence on alias analysis

In the explanations given so far, we omitted an important problem. PHP allows the creation of aliases, which means that two or more variables can point to the same memory location. If one of these variables is assigned a new value, it also affects the aliases of the variable. Ignoring this issue would prevent literal analysis from producing correct results in a number of cases. The following Section 7 provides a more detailed introduction to PHP references and describes an alias analysis that collects the alias information required by literal analysis. Section 8 revisits literal analysis when alias information is taken into account, and presents transfer functions for the remaining CFG nodes.

Note that in our current approach, we employ literal analysis only for resolving file inclusions (Section 10). Judging from the experiences we made during our empirical evaluation, the names used in file inclusion operations never depend on aliasing in practice. This is why our literal analysis does not make use of the fully-fledged alias analysis that will be described in the next section, resulting in a performance boost without loss of precision. In spite of this fact, we will describe how literal analysis can benefit from alias analysis. One reason is that we believe this description to be useful for other researchers. The other reason is that the underlying concepts are analogous to those necessary for feeding alias information into taint analysis (Section 9).

7. Alias analysis

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.

7.1. 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 7 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.

In the past, extensive work has been devoted to the area of alias analysis (e.g., [2, 6,24,39,45], to mention only a few). An overview of existing solutions and open issues is given by Hind in [14]. During our investigations on how to solve the problem of computing alias information for PHP programs, we were considering to use one of these existing analyses. However, many of the existing approaches had some particular limitation that made sense in the context of C programs, but which we were not willing to accept for PHP programs. For instance, the well-known analyses of Andersen [2], Steensgaard [39] and Das [9] are flow-insensitive. While this considerably speeds up the analysis, the generated results are less precise than with a flow-sensitive analysis, which can lead to false positives. Apart from their mere

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

Fig. 7. Simple aliasing in PHP.

existence, a major problem with such false positives is that it can be difficult to determine their cause (i.e., flow-insensitivity) during the manual inspection of the generated vulnerability reports.

Another problem with reusing existing approaches is that there seems to be no straightforward translation of alias analysis techniques designed for C to a technique that can be used for PHP programs. One reason for the current absence of such a translation is that there are semantic differences between references and pointers. The PHP manual [32] 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* [32] that do not directly address memory locations. That is, alias relationships between variables in PHP are represented internally through equivalence in the context of symbol tables, and not through identical pointer values (i.e., memory addresses). Besides, PHP is dynamically typed, which means that the types of variables are not declared explicitly in the program's source code. Instead, the types of variables are decided at runtime, and can change during program execution. Moreover, PHP does not provide a separate data type for references. Instead, *all* variables are references by nature, even those containing only scalar values.

Figure 8 illustrates another difference, which occurs in combination with parameter passing. 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. Also note that PHP references are mutable, as opposed to references in C++.

To the best of our knowledge, the issues discussed above have not been addressed in the literature so far. Minamide [27] and Huang et al. [16] briefly mention their use of alias analysis for PHP, but without providing details. Similarly, Liu et al. [25] only give a short mention that they have applied existing pointer analysis algorithms to Python programs.

```
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 }
```

Fig. 8. References in contrast to pointers.

7.2. Intraprocedural alias analysis

Figure 9 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. 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 the Appendix). Figure 28 in the Appendix 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 complete graphs). Note that this combination operator does not simply compute must-aliases through intersection and may-aliases 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 Fig. 9 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.

```
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)}
```

Fig. 9. Intraprocedural analysis information.

7.3. 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.

7.4. 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 [30] (i.e., it would have an infinite height), the analysis would not terminate in such cases. Our solution to this problem (as first presented in [21]) 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 important rule leads to a finite number of variables during the analysis and forms the basis for our alias analysis approach.

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 first give a brief 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.

The aliases between the caller's local variables cannot be modified by the callee. Note that this does not imply that the *values* of the caller's locals cannot be modified by the callee (but such changes of values are not relevant for alias analysis). Similarly, the aliases between the callee's local variables are always the same on function entry.

The alias relationships listed above cover all possible cases that can occur in an application. Thus, they represent a complete partitioning of the problem space, making our approach sound with regard to the currently supported language constructs (i.e., modulo the limitations discussed in Section 9.4). 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 the Appendix.

7.4.1. 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 was 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 [38]. This means that alias information about global variables is propagated verbatim into the callee. Inside the callee, this information can be modified analogously to the modification of local aliases. Finally, the (perhaps modified) global alias information is propagated back to the caller.

An example for the handling of global variables is given in Fig. 10. 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

```
 1 skip;  // u{} a{}
 2 a();
 3 skip;  // u{(m.x1, m.x2, m.x3)} a{}
 4
 5 function a() {  // u{} a{}
 6   $a1 =& $a2;   // u{(a.a1,a.a2)} a{}
 7   $GLOBALS['x1'] =& $GLOBALS['x2'];
 8   skip;      // u{(a.a1,a.a2) (m.x1, m.x2)} a{}
 9   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 }
```

Fig. 10. Aliases between global variables.

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.

7.4.2. 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 Fig. 11 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.

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 3 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

```
 1 a();
 2
 3 function a() {   // u{} a{}
 4   $a1 =& $a2;    // u{(a.a1, a.a2)} a{}
 5   b(&$a1, &$a2);
 6 }
 7
 8 function b(&$bp1, &$bp2) {
 9   skip;   // u{(b.bp1, b.bp2)} a{}
10 }
```

Fig. 11. Must-aliases between formal parameters.

Table 3
May-aliases between formal parameters resulting from calls to a function with signature b(&bp1, &bp2)

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)

the case in the second row of Table 3 is shown in Fig. 12. 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 29 in the Appendix shows the exact algorithm that was applied in this case.

7.4.3. 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.
- It is a global variable (and hence, a trivial must-alias of a global variable).

```
 1 a();
 2
 3 function a() {  // u{} a{}
 4   if (...) {
 5     $a1 =& $a2; // u{(a.a1,a.a2)} a{}
 6   }
 7   skip;         // u{} a{(a.a1,a.a2)}
 8   b(&$a1, &$a2);
 9 }
10
11 function b(&$bp1, &$bp2) {
12   skip;       // u{} a{(b.bp1,b.bp2)}
13 }
```

Fig. 12. May-aliases between formal parameters.

```
 1 a();
 2
 3 function a() {           // u{} a{}
 4   $a1 =& $GLOBALS['x1']; // u{(a.a1,m.x1)} a{}
 5   b(&$a1);
 6 }
 7
 8 function b(&$bp1) {    // u{(m.x1,b.bp1)} a{}
 9   skip;
10 }
```

Fig. 13. Must-aliases between formal parameters and global variables.

- 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 13 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.

7.4.4. 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. That is, the alias information of other global variables can be changed such that these other global variables are now aliasing the aforementioned global variable (and thus, the local variable as well).
   (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.

To illustrate the value of shadow variables, consider Fig. 14, which shows a code snippet covered by Case 1b. At the time of the call to function "b" on Line 10, the local variable $a1 is a must-alias of the global variable $x1. Inside the called function on Line 18, 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 27 in the Appendix. The interested reader is referred to our web site [34] 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.

```
 1 a();
 2 skip;   // u{(m.x1, m.x2)} a{}
 3
 4 function a() {  // u{(m.x1, a.x1_gs)
 5                 //   (m.x2, a.x2_gs)} a{}
 6
 7   $a1 =& $GLOBALS['x1'];
 8   skip;  // u{(m.x1, a.x1_gs, a.a1)
 9          //   (m.x2, a.x2_gs)} a{}
10   b();
11   skip;  // u{(m.x1, m.x2, a.x2_gs)
12          //   (a.a1, a.x1_gs)} a{}
13 }
14
15 function b() {   // u{(m.x1, b.x1_gs)
16                  //   (m.x2, b.x2_gs)} a{}
17
18   $GLOBALS['x1'] =& $GLOBALS['x2'];
19
20   skip;   // u{(m.x2, b.x2_gs, m.x1)} a{}
21 }
```

Fig. 14. Aliases between local variables and global variables.

7.5. Complexity

Since we have integrated our alias analysis into a standard data flow analysis framework, the worst case for the number of basic operations (least upper bound operations and transfer function applications) applied by the underlying iterative algorithm is O($n \cdot h$), where n denotes the number of CFG nodes, and h denotes the height of the used lattice [30]. In this rare worst case, the analysis information for every node in the program moves only one step up in the lattice for each pass of the algorithm, and eventually reaches the top element. In our alias analysis, the top element (i.e., the element that holds the least precise information) represents the information that every variable in the program is a may-alias of every other variable. In this top element, the number of may-alias pairs equals $\sum_{i=1}^{v-1} i = \frac{(v-1)\cdot v}{2}$, where v denotes the number of variables in the program. Note that the actual number of variables that the analysis operates on (v) is higher than the number of variables in the scanned program (V), due to the requirement of shadow variables. In the worst case, the program has only global variables, which results in $v = 2 \cdot V$. The precision of the information of the top lattice element can be improved by removing one alias pair after the other, which is equivalent to moving down the lattice step by step, until the empty lattice element is reached. As a result, the height of the lattice is identical to the number of may-alias pairs in the top lattice element, leading to a quadratic number O($n \cdot v^2$) of basic operations in the worst case. The basic operations are linear in the size of their input elements, and hence, do not change the quadratic runtime behavior.

Regarding space requirements, the worst case occurs if every CFG node is associated with the largest possible lattice element. Thus, the worst-case space complexity is O($n \cdot s$), where s denotes the maximum lattice element size. The largest element is the top lattice element, which requires $\frac{(v-1)\cdot v}{2}$ space, leading to a worst case space complexity of O($n \cdot v^2$). Note that our empirical results indicate that the practical time and space requirements are typically significantly lower than in the worst case, since complex alias relationships are rare in real-world PHP programs.

8. Literal analysis revisited

Using the information collected by alias analysis, we can extend the basic concepts for literal analysis presented in Section 6 to correctly include alias relationships. For a simple assignment of the form "\$a = \$b", the aliases of the variable on the left side are relevant for literal analysis. The reason is that all aliases of variable \$a are also affected by the assignment. Of the four cases that literal analysis had to distinguish in Section 6, all but the first involved an array or array element on the left side. Since we only consider references to simple variables, there cannot exist aliases for the variable on the left side in these cases. Hence, no further extensions are necessary. In the first case, however, the left variable is not an array element (and not known as

array), and therefore, might possess aliases. Must-aliases of this variable are treated by the transfer function with a strong update, i.e., their literals are simply overwritten with the literal read from the right side of the assignment. As there is no sufficient certainty for may-aliases, they must be handled conservatively by a weak update. That is, all aliases of the variable on the left-hand side are assigned the least upper bound of their literal and the literal from the right side. An overview of the possible cases and actions for simple assignment nodes is given in Table 2.

Now that the transfer function definition for simple assignment nodes is completed, the transfer functions for the other relevant CFG nodes can be introduced. For *unary assignment nodes* such as "$a = −$b", the literal resulting from the application of the operator on the right side is computed first, and then the presented technique for simple assignment nodes is used. For example, if $b evaluates to 3, $a is assigned −3. Special care was taken to reflect PHP's implicit type conversion mechanisms, which are beyond the scope of this paper. The treatment of *binary assignment nodes* such as "$a = $b + $c" is analogous to unary assignments, with the sole difference that the operator on the right side takes two input arguments instead of one. Note that when we do not dispose of specific information for an operand (i.e., if it is $\top$), the result of the operation is also $\top$. In essence, the actions performed by the transfer functions introduced above correspond to the classic combination of constant propagation and constant folding.

For *reference assignment nodes* such as "$a =& $b", it is sufficient to overwrite the literal of $a with the literal of $b, since reference statements have been restricted to simple variables. *Global nodes* can be handled as normal assignments, with the operand variable on the left side and an equally-named variable from the global scope on the right side. In the case of *unset nodes*, the unset variable and all its literal array elements are set to the NULL literal, which represents the PHP null value. As far as literal analysis is concerned, *array assignment nodes* have the same semantic effects as unset nodes, and can be treated in the same way.

What remains to be specified are the interprocedural transfer functions. At *call preparation nodes*, we iterate through each formal parameter of the callee. If there exists a corresponding actual parameter, the analysis captures the value transfer by simulating an assignment with the shape "formal parameter = actual parameter". When the programmer specifies default values for formal parameters, an invocation does not have to provide actual parameters for these. For example, a function with the signature "foo($p = 7)" does not have to be called with any actual parameters at all. In such cases, the analysis simply performs assignments with the shape "formal parameter = default value". Afterwards, local variables are reset to their initial values (i.e., those values that the variables were assigned at the very beginning of the analysis). The reason is that, analogous to the interprocedural rule given in Section 7.4, values of local variable incarnations must only be tracked inside the function that these variables belong to.

At the *call return node*, the analysis must make sure that both global and local variables of the caller receive correct values in accordance to the effects that the

callee had on them. For global variables, this is straightforward: We simply propagate the values from the end of the callee back to the caller (as done by Sharir and Pnueli in [38]). For local variables, the situation is more difficult. There are two ways how a callee can affect the value of a caller's local variable:

- If a local variable of the caller was aliased with a global variable at call-time, the callee can modify this local variable by modifying the global variable.
- If a local variable of the caller was used as actual call-by-reference parameter at call-time, the callee can modify this local variable by modifying the corresponding formal parameter.

Note that it is necessary to be aware of the possibility that the aliased global or formal variable could be redirected to some other variable inside the callee. Hence, it is not safe to simply assign the value of the global or formal to the local variable. Instead, we have to make use of the already introduced shadow variables. Remember that shadow variables serve as representatives of local variables involved in alias relationships with globals or formals. As a result, they are perfectly suited for the task at hand. For both of the above cases, we must further distinguish whether the local variable was involved in a must- or a may-alias relation. We will now discuss the treatment of each of the four resulting cases in detail.

Global must-aliases. For each local variable of the caller that has a global must-alias at the time of the function call, we choose one of these globals (it does not matter which one we choose, as they must all hold the same literal value). In accordance to our previous remark about the usage of shadow variables, the analysis now sets the literal of the local variable to the literal of the corresponding g-shadow. Then, the local variable is marked as having been visited, informing the following steps that it is not necessary to perform any further computations for this variable's literal. An example for this computation step is given in Fig. 15. When reaching the call to function "b" on Line 7, the local variable $a1 of function "a" has a must-alias relation with global variable $x1. Hence, function "b" has the power to modify the value $a indirectly by modifying $x1, and does so on Line 12. On Line 13, $x1 is redirected away from $a1 and the g-shadow $x1_gs to another global variable ($x2). This is the reason why the assignment on Line 15 does *not* affect $a1, which is reflected by $x1_gs keeping its old value. When the function returns, we simply set $a1 to the value of $x1_gs, achieving the desired result.

Formal must-aliases. For each local variable that was used as actual call-by-reference parameter at call-time, the literal is set to that of the corresponding f-shadow (which is analogous to what we did in the previous step). In addition, we do the same with all local must-aliases of these local variables, since the local must-aliases are also must-aliases of the corresponding formal parameters. Note that we skip local variables that have already been marked as visited, and mark those local variables that are processed. On Line 5 in Fig. 16, local variables $a1 and $a2 of

```
 1 $x1 = 1;
 2 $x2 = 2;
 3 a();
 4
 5 function a() {
 6   $a1 =& $GLOBALS['x1']; // a1:1, x1:1
 7   b();
 8   skip;                  // a1:7, x1:8
 9 }
10
11 function b() {
12   $GLOBALS['x1'] = 7; // x1:7, x1_gs:7
13   $GLOBALS['x1'] =&
14     $GLOBALS['x2'];   // x1:2, x1_gs:7
15   $GLOBALS['x1'] = 8; // x2:8, x1_gs:7
16 }
```

Fig. 15. Modification of caller locals due to a global must-alias.

```
 1 a();
 2
 3 function a() {
 4   $a1 = 1;     // a1:1
 5   $a2 =& $a1;  // a1:1, a2:1
 6   b(&$a1);
 7   skip;        // a1:7, a2:7
 8 }
 9
10 function b(&$bp1) { // bp1:1, bp1_fs:1
11   $bp1 = 7;          // bp1:7, bp1_fs:7
12   $b2 = 2;     // bp1:7, bp1_fs:7, b2:2
13   $bp1 =& $b2; // bp1:2, bp1_fs:7, b2:2
14   $bp1 = 8;    // bp1:8, bp1_fs:7, b2:8
15 }
```

Fig. 16. Modification of caller locals due to a formal must-alias.

function "a" are aliased. On the next line, function "b" is called with $a1 as actual call-by-reference parameter. Hence, the assignment to the corresponding formal parameter $bp1 on Line 11 affects both $a1 and $a2. The assignment on Line 14 has no effect on these two local variables since $bp1 was redirected to $b2 on Line 13. Again, the use of a shadow variable (in this case, the f-shadow $bp1_fs) has enabled the algorithm to deal with this situation correctly.

Global may-aliases. Now, we turn our attention towards the modification of local variables enabled by the presence of may-alias relationships. For each local variable that was a may-alias of a global variable at call-time, its literal is set to the least upper bound of its literal at call-time and the literal of the corresponding g-shadow. As before, we can skip local variables that have been marked as visited. However, we do not mark any variables as visited in this step, since the flow of values due to may-aliases is not definitive and might be changed by the following step. When control flow reaches the call on Line 8 of Fig. 17, there exists a may-alias relationship between $a1 and $x1. Inside function "b", the value of $x1 is changed from 1 to 7. Since $a1 is only a may-alias of $x1, we have to compute the least upper bound of 1 and 7 instead of setting the value of $a to 7, resulting in a1:$\top$ on Line 10.

Formal may-aliases. A local variable is a may-alias of a formal parameter if it is a may-alias of an actual call-by-reference parameter at call-time. The case where this actual parameter is a global variable can be skipped here, since it has already been handled by the previous step. Hence, we only focus on local variables that are may-aliases of local actual call-by-reference parameters. For each of these variables that has not been marked as visited, we compute its literal value as the least upper bound of its literal value at call-time and that of the corresponding f-shadow. When control flow reaches the call on Line 7 of Fig. 18, there exists a may-alias relation between $a1 and $a2. While the assignment to $bp1 on Line 13 directly overwrites the value of the actual call-by-reference parameter $a1, computing the least upper bound is necessary for the may-alias $a2, resulting in a2:$\top$ on Line 9.

Finally, the return value of the callee is passed back to the caller by assigning it to the temporary variable provided by the P-Tac conversion for representing the function's expression value.

```
 1 $x1 = 1;
 2 a();
 3
 4 function a() {
 5   $a1 = 1;
 6   if (*)
 7     $a1 =& $GLOBALS['x1']; // a1:1
 8   skip;                    // a1:1
 9   b();
10   skip;                    // a1:⊤
11 }
12
13 function b() {         // x1:1, x1_gs:1
14   $GLOBALS['x1'] = 7; // x1:7, x1_gs:7
15 }
```

Fig. 17. Modification of caller locals due to a global may-alias.

```
 1 a();
 2
 3 function a() {
 4   $a1 = $a2 = 1; // a1:1, $a2:1
 5   if (*)
 6     $a2 =& $a1;  // a1:1, $a2:1
 7   skip;          // a1:1, $a2:1
 8   b(&$a1);
 9   skip;          // a1:7, a2:⊤
10 }
11
12 function b(&$bp1) { // bp1:1, bp1_fs:1
13   $bp1 = 7;         // bp1:7, bp1_fs:7
14 }
```

Fig. 18. Modification of caller locals due to a formal may-alias.

Functions that are built into PHP are conservatively modeled as returning $\top$, since the increased precision is expected to be rather small compared to the required work necessary for simulating these functions. For a safe approach, the analysis can additionally set all reference parameters to $\top$ as well. The only built-in function that is modeled precisely is the frequently used *define*, which is needed for the definition of constants.

9. Taint analysis

With the information from alias analysis at its disposal, taint analysis is finally able to produce satisfactory results (in analogy to the explanations from the previous section). Taint analysis strongly resembles literal analysis. Its purpose is to determine, for each program point, the taint value (instead of the literal) of a variable or constant. Once these results have been computed, it is possible to inspect whether any sensitive sink in the program is receiving malicious data, and hence, to detect vulnerabilities.

9.1. Carrier lattice definition

A variable or constant is said to be tainted if it can hold a malicious,[4] not yet sanitized (checked) value originating from user input. Since literals can never hold user input, they are always untainted. The basic lattice for taint analysis resembles that of literal analysis, with the difference that it does not map to literals and $\top$,

[4]Malicious with respect to cross-site scripting or SQL injection for our prototype implementation, but the presented concepts are able to cover also other forms of taint-style vulnerabilities.

but to the taint values *tainted* and *untainted*. Note that we take a conservative (safe) approach in that a variable being mapped to tainted means "this variable *might* be tainted", whereas a mapping to untainted means "this variable *is* untainted". Hence, whenever the analysis cannot determine whether a variable is tainted or not, it is conservatively assumed to be tainted. In the context of data flow analysis, being tainted is therefore less precise than being untainted, which is illustrated by the lattice fragment in Fig. 19. Just like for the previous analyses, less precise lattice elements are located above more precise elements. In practice, the lattice elements do not only contain a mapping for one single variable, but for all variables that occur in the program. This lattice corresponds to the typical taint lattice that has already been used by other researchers [16,36].

Recall that literal analysis treats non-literal array elements (such as $a[$i]) in a pessimistic way in that they are always mapped to $\top$. For the same reasons, taint analysis maps non-literal array elements to tainted. In the context of taint analysis, however, this leads to undesirable false positives in certain cases. For example, declaring a variable with the built-in "array" function clears all its content, including all values possibly injected by an attacker (see Fig. 20). That is, the "array" function is considered to belong to the built-in sanitization functions of PHP. Whenever an array is cleared in such a manner, we would like to treat its content as untainted, even though taint analysis might yield a different result. This is why we track an additional *clean array flag* (CA flag) for each variable that is not an array element. An active CA flag overrides the taint information for the whole array tree, meaning that all its elements (including those with non-literal indices) are untainted. The CA flag of an array element is implicitly considered to be equal to the CA flag of its enclosing array.

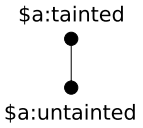

Fig. 19. Fragment of a taint analysis lattice.

```
1 $a = <user input>;
2 // $a[1] can be controlled by an attacker
3
4 $a = array();
5 // now $a[1] is no longer controlled
6 // by an attacker
```

Fig. 20. Untainting with "array".

9.2. Transfer functions definition

Due to its similarity to literal analysis, the transfer functions for taint analysis will sound familiar. For the *simple assignment node*, the same distinction of cases with regard to the left variable has to be made, and the operations affecting the taint values are completely analogous to those for literal values. The only difference is that taint analysis also has to consider the "CA flag" extension to its lattice. An overview of the necessary operations is given in the rightmost column of Table 2. The "root" field used in Table 2 denotes the variable itself if it is not an array element (e.g., $a.root == $a) and the root node of the corresponding array tree otherwise (e.g., $b[1][2].root == $b).

As mentioned in Section 8, special care was taken to handle PHP's implicit type conversion mechanisms for *unary assignment nodes*. Doing this in the context of taint analysis has the desirable effect that sanitization through type casting is handled correctly. For instance, implicitly casting a tainted variable into an integer (with unary operators such as $+$, $-$ and (int)) untaints this variable, since attacks typically involve more data (either scripts to display to a user or SQL code to alter the meaning of a query) than just simple integers in order to work properly. The same holds for *binary assignment nodes*.

For *reference assignment nodes* such as "$a =& $b", we can adapt the literal analysis transfer function in a straightforward fashion: It is sufficient to overwrite the taint value and CA flag of $a with the taint value and CA flag of $b. At *unset nodes*, the operand variable and all literal array elements are set to untainted. If the operand variable is not an array element, its CA flag is set to "clean" (recall that CA flags are not tracked explicitly for array elements). *Array assignment nodes* and *global nodes* are treated as usual, i.e., analogous to unset nodes and reference assignment nodes, respectively.

The interprocedural operations necessary for taint analysis are completely analogous to literal analysis, with the differences for handling CA flags already discussed.

In contrast to literal analysis, it is important for taint analysis to correctly model built-in PHP functions in order to reduce the number of false positives. For this purpose, Pixy processes a specification file on startup which contains (currently more than 50) abstracted versions of built-in functions in PHP syntax. For modeling taint values, the special placeholder variables $_TAINTED and $_UNTAINTED are used. For instance, the effect of the built-in function "htmlentities", which is effective in sanitizing input against cross-site scripting attacks, is implemented by simply letting it return $_UNTAINTED. Similar considerations apply to functions that sanitize input against SQL injection attacks, such as "mysql_escape_string".

9.3. Using the analysis results

Generating warnings that point the developer to possible cross-site scripting vulnerabilities at the end of the analysis is straightforward. The analysis information

for each sensitive sink (such as calls to "echo" and "mysql_query") is searched for tainted input variables, and a warning message indicating the corresponding line is issued if such a violation is discovered.

9.4. Limitations

Currently, Pixy does not support object-oriented features of PHP. Each use of object member variables and methods is treated in an optimistic way, meaning that malicious data can never arise from such constructs. In alias analysis, object or member variables never appear as elements of alias relationships. Besides, reference statements that contain arrays or array elements are not considered by alias analysis. However, this restriction did not appear to impact 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.

10. 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 inclusion 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.

11. Empirical results

We performed a series of experiments with our prototype implementation to demonstrate its ability to detect previously unknown XSS and SQLI vulnerabilities. To this end, Pixy was run on the current versions of seven popular, open-source PHP programs (four of which make use of a backend database). 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 interacting with the web application. We provided these entry points as input files to Pixy, which automatically resolved further file inclusions.

Table 4 shows a summary of our results, including the number of entry points and the total lines of code that were analyzed. 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 1 GB RAM, even though our prototype still presents many

Table 4
Summary of vulnerability reports

Program	Entry files	LOC	Time (s/file)	Vulnerabilities		False positives	
				XSS	SQLI	XSS	SQLI
DCP Portal 6.1.1	22	61,617	6.0	76	83	15	14
MyBloggie 2.1.3beta	6	20,326	58.3	13	31	3	11
OSIC 0.7	14	14,281	1.5	42	65	12	1
NewsPro 1.1.4	22	4,251	1.1	4	14	14	34
QaTraq 6.5	146	4,275,234	10.7	48	–	14	–
Qdig 1.2.9.2	4	5,385	54.5	2	–	14	–
TxtForum 1.0.4-dev	15	4,398	1.3	31	–	17	–
Totals	229	4,385,492	10.2	216	193	89	60

opportunities for performance tuning. There was no analysis run that took longer than five minutes for a single entry point.

11.1. XSS detection

In total, we discovered 216 exploitable XSS vulnerabilities in the programs that we analyzed. 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 40% is relatively low and further alleviated by the fact that many false positives are similar, which makes their recognition easier (see Section 11.1.2 for more details). 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.

Note that for QaTraq, there were actually 209 additional reports not listed in Table 4. The programmers of QaTraq frequently reused large parts of code through copy-and-paste. As a result, the places that these additional reports refer to strongly resemble those for which we constructed working exploits. Even though we are confident that these reports correspond to real vulnerabilities, we did not bother to write exploits for them (due to a certain repetitiveness that would have been connected with this task). To remain sincere, we abstained from listing these reports as vulnerabilities. Note that copy-and-paste programming is also the reason for the relatively high number of entry files and lines of code for QaTraq.

11.1.1. A case study: MyBloggie

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 would be quite difficult.

Figure 21 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,

```
 1 if (...) {
 2   multi_tb($post_urls, ...);
 3   $tbstatus = $tbstatus . $tbreply;
 4 } else {
 5   $tbstatus ="";
 6 }
 7
 8 message(..., $tbstatus);
 9
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 }
```

Fig. 21. Vulnerability in MyBloggie (simplified).

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 7.3, 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.

11.1.2. False positives

A majority of the reported false positives (62 of 89) were due to impossible program paths. Figure 22 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.

```
 1 while ($row = mysql_fetch_array($result)) {
 2   $site_name = $row["site_name"];
 3 }
 4 echo SelectMember(..., ...)
 5
 6 function SelectMember($id, $opt) {
 7   global $site_name;
 8   if (...) {
 9     ...
10   } else {
11    return $site_name;
12   }
13 }
```

Fig. 22. False positive due to impossible path (simplified).

The remaining 27 false positives fall into various categories. Eight reports were caused by an unusual piece of code in Qdig for which our modeling of the PHP `explode` function was not precise enough. The `explode` function takes two string parameters, $sep and $string, and returns all substrings of $string that are separated by $sep. Our current model of this function does not perform a detailed inspection of its parameters, but conservatively approximates its behavior by returning the taint status of $string. Six reports were caused by a conceptual limitation of the call string technique [38], which is used for interprocedural data flow analysis. This number could be reduced by increasing a numeric analysis parameter (the "k-bound") that represents analysis precision. The trade-off would be a negative impact on performance. In four cases, an `if` construct in the program was syntactically incomplete (i.e., it had no `else` branch), but semantically complete. Four times, a variable was validated using regular expression functions, and another four times, a variable was set dynamically by reading its name from a database. Finally, in one case, a variable was set dynamically using PHP's `eval` function.

11.2. SQL detection

As shown in Table 4, Pixy detected 193 SQLI vulnerabilities and additionally reported 60 false positives (which results in a false positive rate of 24%). Three of our tested applications (DCP Portal, MyBloggie, and NewsPro) were also tested by Xie and Aiken for SQL injection in [46]. By comparing our results to their advisory [35], we found that all vulnerabilities that they discovered are also reported by our system. In addition, we discovered several vulnerabilities that were not among Xie and Aiken's reports. This indicates that our prototype computes more comprehensive results, at least for the tested applications.

```
 1 $file_id = getVAR("id");
 2 $catalogue_id = getValueFromID($file_id);
 3
 4 function getVAR($nm) {
 5   $tmp = "";
 6   if (isset($HTTP_GET_VARS[$nm]))
 7     $tmp = $HTTP_GET_VARS[$nm];
 8   if (isset($HTTP_POST_VARS[$nm]))
 9     $tmp = $HTTP_POST_VARS[$nm];
10   return $tmp;
11 }
12
13 function getValueFromID($id) {
14   $result = mysql_query("
15    SELECT * FROM sometable WHERE ID=$id");
16 }
```

Fig. 23. SQLI vulnerability from OSIC.

11.2.1. A case study: Open searchable image catalogue (OSIC)

Figure 23 shows an SQLI vulnerability from OSIC. In this example, the variable `$file_id` is retrieved from a user-supplied value by calling the function `getVAR` on Line 1. Then, this value is passed to the function `getValueFromID` on Line 2, where it leads to an SQLI vulnerability (Line 14).

11.2.2. False positives

Of the total of 60 false positives that we observed, we found that 31 of them (52%) are due to impossible program paths. Figure 24 shows a typical example. At first glance, the query statement on Line 3 appears to be an obvious SQLI vulnerability, since it directly contains a user-provided POST variable. However, this value is *implicitly validated* in the preceding two lines. The potentially vulnerable query statement is only reached during runtime if the POST variable is equal to the name of a file in the "users" directory. Further examination revealed that certain checks in the program prevent the creation of files with arbitrary names. Hence, given that the attacker injects a malicious value for the POST variable, the application can never take the program path that corresponds to the supposed vulnerability. As a result, the code in Fig. 24 is not vulnerable.

The false positives in the second largest group (25 or 42%) were due to validation using regular expression functions. All of these 25 reports were issued for NewsPro, and an example is given in Fig. 25. On Line 1, it is checked whether the input stored in variable `$newsid` matches a regular expression that corresponds to a non-empty sequence of digits. If this match fails, the program exits. Thus, there is no possibility for an attacker to inject malicious characters into the query on Line 5. However, Pixy

```
1 $lower = chop(strtolower($_POST[reg_uname]));
2 if (file_exists("users/$lower.php")) {
3   mysql_query(...$_POST[reg_uname]...);
4   die();
5 }
```

Fig. 24. Impossible path.

```
1 if (!ereg('^[0-9]+$', $newsid)) {
2   unp_msgBox($gp_invalidrequest);
3   exit;
4 }
5 $check = $DB->query("SELECT * FROM `unp_news`
6   WHERE newsid='$newsid'");
```

Fig. 25. Custom validation from NewsPro.

still considers the variable as potentially dangerous on Line 5 and issues a corresponding alert.

In three additional cases of false positives, a variable was set dynamically by reading its name from a database table, and in one case, a custom regular expression was applied for sanitization.

11.3. File inclusion effectiveness

Table 5 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. 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 frequently occur. As a result, there is a 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 Fig. 26. In this example, it is impossible to include a remote file

Table 5
Summary of file inclusions (average numbers)

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	0.0
OSIC 0.7	1.0	2.8	0.0	0.0
NewsPro 1.1.4	1.0	4.2	0.0	0.0
QaTraq 6.5	1.0	125.2	0.0	0.0
Qdig 1.2.9.2	1.0	0.0	0.0	0.0
TxtForum 1.0.4-dev	1.5	1.8	2.6	1.7

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

Fig. 26. A harmless and a dangerous unresolvable inclusion.

(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 Fig. 26 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.

12. Related work

There exist a number of approaches that deal with static detection of web application vulnerabilities. Huang et al. [16] 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 [17]. A substantial fraction of PHP files (8% in their experiments) is rejected due to problems with the applied parser. In contrast, we are able to parse the full PHP language. Moreover, Huang et al.'s work leaves out important issues such as the handling of references, array elements, file inclusions,

or any of the limitations that we addressed in Section 9.4. Unfortunately, comparing their results to ours was not possible due to the limited detail of their reports (no version numbers or advisory ID's are given).

In [13], the authors applied the Java String Analyzer by Christensen et al. [7] to extract models of a program's database queries, and used these models as the basis for a runtime monitoring and protection component for SQL injection attacks. The main difference compared to our approach is that the extracted models do not contain information about the taint status of embedded variables. As a result, it is not possible to detect vulnerabilities using static analysis only. In our system, we can identify vulnerabilities using static analysis alone.

Work by Xie and Aiken [46] 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 [26] applied an analysis supported by binary decision diagrams developed by Whaley and Lam [43] 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.

An analysis system to detect SQL injection vulnerabilities was recently introduced by Wassermann and Su [42]. For their concurrent work, the authors developed a static analysis technique to determine the possible string values of variables in PHP programs. This allows the analysis of SQL queries and the detection of possible injection vulnerabilities.

Minamide [27] 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 notes 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.

Engler et al. have published various static analysis approaches to finding vulnerabilities and programming bugs in C programs. In [10], 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 [11]. Finally, tainting analysis is used to identify vulnerabilities in oper-

ating 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 [12], which has been, among other things, used by Shankar et al. [36] to detect format string vulnerabilities in C code. However, it remains an open question whether this technique can be applied to untyped scripting languages.

13. 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. Therefore, we propose a static analysis technique that is able to detect taint-style vulnerabilities automatically. This broad class includes many types of common vulnerabilities such as SQL injection or cross-site scripting. Our analysis is based on data flow analysis, a well-understood and established technique in computer science. To improve the correctness and precision of our taint analysis, we conduct a supplementary alias analysis using shadow variables. This alias analysis is specifically targeted at the reference semantics of PHP and generates precise results even for conceptually difficult aliasing problems. Moreover, we presented an iterative, two-stage preprocessing step based on literal analysis for the automatic resolution of file inclusions. All our analyses are interprocedural, context-sensitive and flow-sensitive for providing a high degree of precision and keeping the number of false positives low, making our tool useful for real-world applications.

We tested our concepts by running Pixy, our open-source prototype implementation, on seven open-source PHP 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 provide an effective solution to this problem, offering benefits to both users and providers of web applications.

Acknowledgments

This work was supported by the Austrian Science Foundation (FWF) under grant P18764 (Web-Defense), and by the Secure Business Austria competence center.

Appendix: Algorithms

```
- 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
- foreach 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)
    - foreach 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
- foreach may-alias pair containing a local and a global
  in origInfo:
  - foreach 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
- foreach 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)
    - foreach 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"
  - foreach local may-alias lma of p:
    - foreach 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
```

Fig. 27. Algorithm for computing the alias information after a function call.

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

Fig. 28. Algorithm for the combination operator.

```
- foreach call-by-reference pair:
  - create a placeholder variable for the formal parameter
    and add it to the actual parameter's must-alias group
  - foreach 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
```

Fig. 29. Algorithm for adjusting the alias information that is propagated into a callee.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison–Wesley, Boston, MA, USA, 1986.

[2] L.O. Andersen, Program analysis and specialization for the C programming language, PhD thesis, University of Copenhagen, 1994.

[3] K. Ashcraft and D. Engler, Using programmer-written compiler extensions to catch security holes, in: *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2002.

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

[5] CERT, CERT Advisory CA-2000-02: Malicious HTML tags embedded in client web requests, 2005, http://www.cert.org/advisories/CA-2000-02.html.

[6] D. Chase, M. Wegman and F.K. Zadeck, Analysis of pointers and structures, in: *PLDI'90: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, NY, USA, 1991.

[7] A. Christensen, A. Møller and M. Schwartzbach, Precise analysis of string expressions, in: *International Static Analysis Symposium (SAS)*, San Diego, CA, USA, 2003.

[8] CUP, CUP: LALR parser generator in Java, 2005, http://www2.cs.tum.edu/projects/cup/.

[9] M. Das, Unification-based pointer analysis with directional assignments, in: *PLDI'00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, BC, Canada, 2000.

[10] D. Engler, B. Chelf, A. Chou and S. Hallem, Checking system rules using system-specific, programmer-written compiler extensions, in: *OSDI 2000*, Denver, CO, USA, 2000.

[11] D. Engler, D.Y. Chen, S. Hallem, A. Chou and B. 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*, Banff, Canada, 2001.

[12] J.S. Foster, M. Faehndrich and A. Aiken, A theory of type qualifiers, in: *PLDI'99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, 1999.

[13] W. Halfond and A. Orso, AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks, in: *International Conference on Automated Software Engineering (ASE)*, Long Beach, CA, USA, 2005.

[14] M. 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*, Snowbird, UT, USA, 2001.

[15] Y.-W. Huang, S.-K. Huang, T.-P. Lin and C.-H. Tsai, Web application security assessment by fault injection and behavior monitoring, in: *WWW'03: Proceedings of the 12th International Conference on World Wide Web*, Budapest, Hungary, 2003.

[16] 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: *WWW'04: Proceedings of the 13th International Conference on World Wide Web*, New York, NY, USA, 2004.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee and S.-Y. Kuo, Verifying web applications using bounded model checking, in: *DSN*, Florence, Italy, 2004.

[18] JFlex, JFlex: The fast scanner generator for Java, 2005, http://jflex.de.

[19] R. Johnson and D. Wagner, Finding user/kernel pointer bugs with type inference, in: *13th USENIX Security Symposium*, San Diego, CA, USA, 2004.

[20] N. Jovanovic, C. Kruegel and E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities (Short paper), in: *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2006.

[21] N. Jovanovic, C. Kruegel and E. Kirda, Precise alias analysis for static detection of web application vulnerabilities. in: *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Ottawa, Canada, 2006.

[22] G.A. Kildall, A unified approach to global program optimization, in: *POPL'73: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Boston, MA, USA, 1973.

[23] E. Kirda, C. Kruegel, G. Vigna and N. Jovanovic, Noxes: A client-side solution for mitigating cross-site scripting attacks, in: *The 21st ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, 2006.

[24] W. Landi and B.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*, San Francisco, CA, USA, 1992.

[25] Y.A. Liu, S.D. Stoller, M. Gorbovitski, T. Rothamel and Y.E. Liu, Incrementalization across object abstraction, in: *OOPSLA'05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, USA, 2005.

[26] V.B. Livshits and M.S. Lam, Finding security errors in Java programs with static analysis, in: *Proceedings of the 14th Usenix Security Symposium*, Baltimore, MD, USA, August 2005.

[27] Y. Minamide, Static approximation of dynamically generated web pages, in: *WWW'05: Proceedings of the 14th International Conference on World Wide Web*, Chiba, Japan, 2005.

[28] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Fransisco, CA, USA, 1997.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley and D. Evans, Automatically hardening web applications using precise tainting, in: *IFIP Security 2005*, Chiba, Japan, 2005.

[30] F. Nielson, H.R. Nielson and C. Hankin, *Principles of Program Analysis*, Springer-Verlag, New York, USA, 1999.

[31] PAG/WWW: Static program analysis, 2005, http://www.program-analysis.com.

[32] PHP: Hypertext preprocessor, 2005, http://www.php.net.

[33] T. Pietraszek and C.V. Berghe, Defending against injection attacks through context-sensitive string evaluation, in: *Recent Advances in Intrusion Detection 2005 (RAID)*, Seattle, WA, USA, 2005.

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

[35] SecurityFocus: 99 potential SQL injection vulnerabilities, 2005, http://www.securityfocus.com/archive/1/419280/30/0/threaded.

[36] U. Shankar, K. Talwar, J.S. Foster and D. Wagner, Detecting format string vulnerabilities with type qualifiers, in: *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, USA, 2001.

[37] S. Shankland, Andreessen: PHP succeeding where Java isn't, 2005, available at: http://www.zdnet.com.au/news/software/soa/Andreessen_PHP_succeeding_where_Java_isn_t/0,2000061733,39218171,00.htm.

[38] M. Sharir and A. Pnueli, *Two Approaches to Interprocedural Data Flow Analysis*, Prentice-Hall, Upper Saddle River, NJ, USA, 1981, Chapter 7.

[39] B. Steensgaard, Points-to analysis in almost linear time, in: *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, USA, 1996.

[40] Z. Su and G. Wassermann, The essence of command injection attacks in web applications, in: *Symposium on Principles of Programming Languages (POPL)*, Charleston, SC, USA, 2006.

[41] L. Wall, T. Christiansen, R.L. Schwartz and S. Potter, *Programming Perl*, 2nd edn, O'Reilly & Associates, Sebastopol, CA, USA, 1996.

[42] G. Wassermann and Z. Su, Sound and precise analysis of web applications for injection vulnerabilities, in: *Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, 2007.

[43] J. Whaley and M.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*, Washington, DC, USA, 2004.

[44] Wikipedia, Hasse diagram, 2005, http://en.wikipedia.org/wiki/Hasse_diagram.

[45] R.P. Wilson and M.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*, La Jolla, CA, USA, 1995.

[46] Y. Xie and A. Aiken, Static detection of security vulnerabilities in scripting languages, in: *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, 2006.