DISSERTATION

Web Application Security

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der Sozial- und Wirtschaftswissenschaften

unter der Leitung von

Privatdozent Dipl.-Ing. Dr.techn. Christopher Krügel
E183-1
Institut für Rechnergestützte Automation

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Mag.rer.soc.oec. Nenad Jovanovic
Matrikelnummer: 0025527
Traiskirchnerstraße 19
A-2512 Tribuswinkel, Österreich

Wien, am 25. Juli 2007

Kurzfassung (German Abstract)

Im Lauf der letzten Jahre hat sich das Web zu einem wesentlichen Bestandteil unseres täglichen Lebens entwickelt. Während unsere Abhängigkeit vom Web zunimmt, steigt gleichzeitig auch das Interesse von Angreifern an der Ausnützung von Sicherheitslücken in Webanwendungen. In dieser Arbeit werden neuartige Ansätze zur Erkennung solcher Sicherheitslücken sowie zum Schutz von Anwendern gegen Webangriffe vorgestellt

Erkennung von Sicherheitslücken. Die prominentesten Arten von Sicherheitslücken in Webanwendungen (wie z.B. SQL Injection und Cross-Site Scripting) gehören zur allgemeinen Klasse der *Taint-basierten Sicherheitslücken.* Die vorliegende Arbeit beschreibt neue Techniken zur Erkennung dieser Art von Sicherheitslücken durch statische Analyse des Quelltexts potentiell verwundbarer Anwendungen. Die vorgestellten Techniken basieren auf flußsensitiver, interprozeduraler und kontextsensitiver Datenflußanalyse, um verwundbare Programmpunkte aufzuspüren. In diesem Zusammenhang werden Algorithmen für die Lösung von Problemen präsentiert, die charakteristisch für die Analyse von Webanwendungen sind.

Schutz von Anwendern. Abgesehen von der proaktiven Erkennung und Behebung von Sicherheitslücken ist es ebenfalls nutzbringend, Echtzeitmethoden zum Schutz von Benutzern von Webanwendungen einzusetzen. Insbesondere Cross-Site Request Forgery ist eine gefährliche Art von Angriff, durch die sich Authentifizierungsmechanismen verwundbarer Anwendungen umgehen lassen. Bestehende Ansätze zur Entschärfung dieser Bedrohung sind unvollständig, zeitaufwendig und fehleranfällig. Wir präsentieren eine proxy-basierte Lösung, die einen zuverlässigen und vollständig automatischen Benutzerschutz für existierende Webanwendungen zur Verfügung stellt. Diese Lösung ist einfach einzusetzen und führt zu keiner Beeinträchtigung des regulären Verhaltens der geschützten Webanwendung.

Die vorgeschlagenen Techniken wurden zur Gänze implementiert und an Beispielen aus der Praxis evaluiert, um deren Machbarkeit, Effektivität und Nützlichkeit zu demonstrieren. Unsere Prototypen wurden unter einer Open-Source-Lizenz veröffentlicht und können von unserer Website [67] heruntergeladen werden.

Abstract

During the last years, the web has evolved into an integral part of our daily lives. Unfortunately, as our dependency on the web increases, so does the interest of attackers in exploiting security vulnerabilities in web applications. This thesis presents novel approaches aimed at the detection of such vulnerabilities, and at the protection of clients against web-based attacks.

Vulnerability Detection. The most prominent types of web application vulnerabilities (such as SQL Injection and Cross-Site Scripting) belong to the general class of *Taint-Style Vulnerabilities*. In this thesis, we describe novel techniques for detecting these types of vulnerabilities by statically analyzing the source code of potentially vulnerable applications. More precisely, our techniques are based on flow-sensitive, interprocedural and context-sensitive data flow analysis to discover vulnerable points in a program. In this context, we present algorithms for the solution of problems unique to the analysis of web applications.

Client Protection. Apart from proactively detecting and fixing vulnerabilities at the server side, it is also beneficial to employ real-time methods for protecting web application users against attacks. In particular, Cross-Site Request Forgery is a dangerous type of attack that is capable of bypassing the authentication mechanism of vulnerable applications. Existing approaches to mitigating this threat are incomplete, time-consuming, and error-prone. We present a proxy-based solution that provides a reliable and fully automatic user protection for existing web applications. Applying this solution is straightforward, and does not interfere with the regular behavior of the protected web application.

The proposed techniques have been implemented and evaluated on real-world examples, demonstrating their feasibility, effectiveness, and usefulness. Our prototype implementations have been released under an open-source license, and are available for download at our web site [67].

Acknowledgements

Professional. I owe my thanks to my advisors, Christopher Kruegel and Engin Kirda, who have safely and skillfully guided me through the entirety of the work presented in this thesis. They have earned my gratitude and respect with their professional competence, their catching enthusiasm, and the admirable ability to keep their students' work steadily on the road of science. Their positive example has strongly influenced my way of working, and has resulted in a thesis that I believe to be both interesting and nice to read.

Familial. Ich danke meinen Eltern für ihre unbeirrbare Zuwendung und Unterstützung, ohne die nicht nur diese Arbeit, sondern auch zahlreiche andere Errungenschaften in meinem Leben nicht möglich gewesen wären. Meinem Bruder danke ich für die vielen gemeinsamen Stunden, aus denen ich so oft und reichhaltig die Kraft für bevorstehende Herausforderungen geschöpft habe. Meine Familie war für mich immer ein Ort der Sicherheit und der gegenseitigen Hilfe, und wird mir als Vorbild für meine weitere familiäre Zukunft dienen.

Emotional. Birgit, meiner geliebten Ehefrau, danke ich für ihre Zuneigung, ihr Interesse, und dafür, daß es sie gibt. Ihr positives Wesen und die Wärme, die sie ausstrahlt, erfüllen mich jedesmal aufs neue mit Glück und frischer Energie. Die vergangenen Jahre, die wir miteinander verbracht haben, waren die schönsten und erfülltesten meines Lebens.

Financial. This work was supported by the Austrian Science Foundation (FWF) under grants P18368 (Omnis) and P18764 (Web-Defense), and by the Secure Business Austria competence center.

Contents

Chapter 1

Introduction

1.1 Motivation

Web applications have become one of the most important communication channels between various kinds of service providers and clients on the Internet. The use of web-based services (such as online shops, news pages, and search engines, to name just a few) has become a wide-spread routine in today's economic and social life. Countless applications are running on millions of servers world-wide, and their numbers are constantly increasing.

Along with the increased importance of web applications in the last years, the negative impact of security flaws in such applications has grown as well. Vulnerabilities that may lead to the compromise of sensitive information are being reported continuously, and the costs of damages resulting from exploited flaws can be enormous. In the past, buffer overflows were the dominating type of vulnerability, and received the highest attention from both the security community and attackers. According to a recent analysis of the CVE [14] archives, buffer overflows have descended to rank number four between 2004 and 2006, whereas the top three ranks are now occupied by web application vulnerabilities [7]. These flaws include common problems such as SQL injection (SQLI) and cross-site scripting (XSS), and pose a serious threat to both providers and users of web-based services. Among the main reasons for this phenomenon are time and financial constraints, limited programming skills, and lack of security awareness on part of the developers. However, even under the best conditions, programmers might make mistakes that introduce security problems into their applications. As a result, the need for automated tools that detect vulnerabilities or protect users against attacks is evident.

1.2 Thesis Overview

The various approaches for mitigating threats to web applications can be divided into client-side and server-side solutions (a more detailed classification and an overview of related work can be found in Chapter 6). In this thesis, we address two main aspects of server-side web application security. First, we present techniques for the static detection of taint-style vulnerabilities. Second, we describe a system for the dynamic protection of web application users against cross-site request forgery attacks. This thesis is based on previous work presented in [35, 36, 37, 38, 39, 40, 6].

1.2.1 Static Detection of Taint-Style Vulnerabilities

Many common web applications vulnerabilities (such as cross-site scripting and SQL injection) share a number of characteristics that makes it possible to regard them as instances of a general vulnerability class. We introduce this class, which we denote as *Taint-Style Vulnerabilities*, in Chapter 2. This classification permits us to focus our concepts on the common properties of these vulnerabilities, and to easily derive solutions for concrete vulnerabilities from the general techniques.

In Chapter 3, 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. We focus our discussion on the detection of cross-site scripting vulnerabilities in PHP applications. PHP is a programming language that is widely used for web application development, and contains several common scripting language features that pose unique challenges to the analysis process. We describe techniques for handling these issues, including untyped arrays, unique referencing semantics, and dynamic file inclusions. Our analysis is designed to achieve a high level of precision, and can be effectively used to detect vulnerabilities in real-world programs.

The techniques presented in Chapter 3 are characterized by a strong focus on the propagation of taint values. In Chapter 4, we advance our previous concepts by *combining taint information with string information*, such that previously undetected vulnerabilities can be discovered. More precisely, we describe a novel analysis for the detection of SQL injection vulnerabilities. This analysis is supported by an automata-based and taint-aware string analysis, and is able to discover even subtle vulnerabilities. During the analysis, taint flow traces are generated, which describe the paths of tainted values through the program, and facilitate the manual inspection of vulnerability reports. We also present a variation of this analysis that is capable of detecting custom forms of input sanitization, which rely on the use of general string-modifying functions to remove harmful properties from user input. Moreover, we demonstrate how to extract *program capabilities* with regard to a program's interaction with the database and the filesystem, and describe how this information can be leveraged to enhance an application's resilience against attacks. To evaluate our techniques in practice, we conducted a comprehensive analysis of seven real-world applications. The empirical results show that our concepts enable us to detect vulnerabilities under a low false positive rate, and that program capabilities can be extracted with a high precision.

1.2.2 Dynamic Prevention of Cross-Site Request Forgery

In contrast to SQL injection and cross-site scripting, cross-site request forgery (XSRF) is a relatively new type of attack that has not received much attention in the literature. In an XSRF attack, the trust of a web application in its authenticated users is exploited by letting the attacker make arbitrary HTTP requests on behalf of a victim user. The problem is that web applications typically act upon such requests without verifying that the performed actions are indeed intentional. Because web application developers are largely unaware of XSRF, there exist many web applications that are vulnerable to XSRF. Unfortunately, existing mitigation approaches are time-consuming and error-prone, as they require considerable manual effort to integrate defense techniques into existing systems. In Chapter 5, we present a solution that provides a completely automatic protection from XSRF attacks. More precisely, our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to users as well as to the web application itself. We provide experimental results to demonstrate that we can use our prototype to secure a number of popular open-source web applications, without negatively affecting their behavior.

Chapter 2

Taint-Style Vulnerabilities

An important observation in the study of web application security is that many types of vulnerabilities (including XSS and SQLI vulnerabilities) belong to the general class of *taint-style vulnerabilities.* The essence of these vulnerabilities is that they share a common “from source to sink” characteristic. In this context, *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 (denoted as *taint sources*), and is propagated through the program via assignments and similar constructs. Using a set of suitable operations, tainted data can be *untainted* (*sanitized*), removing its harmful properties. Apart from XSS and SQLI, other prominent examples for taint-style vulnerabilities include shell injection, script injection, and path traversal attacks. An alternative overview of taint-style vulnerabilities is given by Livshits and Lam in [48]. In this thesis, we have chosen to focus on XSS and SQLI, because these two occur frequently in real-world applications.

2.1 Cross-Site Scripting (XSS)

One of the main purposes of XSS attacks [10] 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. Web browsers are aware of the importance of cookies and hence, they automatically send a cookie only to the web site that created it. Using 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 code 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 malicious input into a database or the filesystem. 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:

- **Taint Sources**: GET, POST, and COOKIE arrays.
- **Sanitization Routines**: PHP functions that delete or alter characters that have a special meaning in JavaScript or HTML code, such as htmlentities(), htmlspecialchars(), and certain type casts (e.g., casts to integer values).
- **Sensitive Sinks**: All routines that return data to the user's browser for display, such as echo(), print() and printf().

2.2 SQL Injection (SQLI)

Most web applications make use of a back-end 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 is present whenever an attacker is able to alter the syntactic structure of such queries in an unexpected way [78]. 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 back-end database, or even compromise the entire host that the web application runs on. Analogous to XSS, it is possible to classify SQLI as an instance of the taint-style vulnerability class that was defined earlier in this chapter:

- **Taint Sources**: GET, POST, and COOKIE arrays.
- **Sanitization Routines**: PHP functions that delete or alter characters that have a special meaning in SQL queries, such as addslashes() and certain type casts (e.g., casts to integer values).
- **Sensitive Sinks**: All routines that pass data to the back-end database, such as mysql_query().

2.3 Shell Injection

The invocation of external programs from within a web application is a popular technique for reusing existing code written in other languages. Otherwise, it would be necessary to re-implement legacy code in the programming language that is used for web application development. Consider the following simple example:

```
$dirname = $_GET['x'];
$result  = system("ls /mypath/${dirname}");
```

In this example, the GET parameter '`x`' is read into the variable $dirname, which is used to assemble a shell command that lists the content of a subdirectory of "/mypath" with the given name. The problem of this code is that it does not consider the possibility that an attacker could use shell meta-characters (i.e., characters that have a special meaning to the shell) to inject arbitrary code to the command shell. For instance, by providing the value "someDir; touch attack.txt" for the '`x`' parameter, the shell command not only lists the content of "someDir", but also creates a file with the name "attack.txt" on the web server. That is, the attacker is able to execute arbitrary shell commands with the rights of the user that owns the web server process. If the attack is not targeted at the invocation of arbitrary commands, but at the malicious alteration of an invoked program's behavior by means of providing additional parameters, it is called *parameter injection.*

2.4 Script Injection

The concept of script injection is analogous to shell injection, and only differs with respect to the language of the injected code. In the case of script injection, the injected code is written in the same (scripting) language as the web application. This type of attack requires that the vulnerable program makes use of routines that interpret their input as scripting code, and execute this code dynamically (e.g., the `eval` function in PHP). If this input is controlled by the attacker and insufficiently sanitized by the application, the attacker can trigger the execution of arbitrary scripting code on the web server.

Due to its unambiguous name, the definition of shell injection is largely agreed upon in the security community. In contrast, script injection is sometimes regarded as "injection of JavaScript code" in the context of XSS attacks. Likewise, the related terms *code injection* and *command injection* are used in an inconsistent way, and potentially include any kind of attack where some type of code is injected. In this sense, these terms also include SQL injection attacks, since SQL code (or commands) are injected into the application.

2.5 Path Traversal (Directory Traversal)

Some web applications assemble the names of filesystem objects (i.e., directories or files) from user input. For instance, the following example code reads and returns the content of a file whose name is retrieved from a user-defined parameter:

```
$filename = $_GET['x'];
$f = file_get_contents("/myDir/$filename");
echo htmlentities($f);
```

This code is vulnerable to a path traversal attack, since it implicitly assumes that the accessed file is located inside directory "/myDir". However, by providing the relative path "../etc/passwd", the system's password file is returned instead.

In cases where the accessed file is used by the program as a source of program statements that are to be executed dynamically, path traversal vulnerabilities can be used to launch script injection attacks. The following code is susceptible to this kind of attack, as it passes unsanitized user input to the file inclusion routine `include`:

```
$filename = $_GET['x'];
include("/myCode/$filename");
```

In order to inject arbitrary scripting code in this example, the adversary is required to create a file that contains the code on the server under attack. This additional obstacle is not present in cases where the name of the included file *begins* with a user-controlled value:

```
$filename = $_GET['x'];
include("${filename}.php");
```

Given that the server is configured to permit remote file inclusions (which is a default setting in the PHP environment), the attacker could place the malicious code at some arbitrary server (e.g., at "http://evilServer/attack.php"), and inject the corresponding URL into the inclusion statement.

2.6 Focus of this Thesis

Even though the explanations and experiments in this thesis are focused on XSS and SQLI vulnerabilities, the presented concepts are targeted at the underlying general properties of taint-style vulnerabilities (i.e., propagation from taint sources to sensitive sinks). As a result, applying the concepts to other types of taint-style vulnerabilities can be reduced to adjusting only a few vulnerability-specific parameters in our implementation.

Chapter 3

Static Vulnerability Detection

In this chapter, we describe techniques for the static detection of XSS vulnerabilities in PHP 4 [59] code by means of data flow analysis. We chose PHP as the target language since it is widely used for creating web applications [77], and a substantial number of security advisories refer to PHP programs [9]. The main contributions described in this chapter 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 necessitates more sophisticated resolution techniques.
- To evaluate our concepts in practice, we implemented Pixy, an open-source web vulnerability scanner. We present empirical results that demonstrate that our tool can be used to detect XSS vulnerabilities in real-world programs. The analysis process is fast, completely automatic, and produces few false positives.

3.1 Data Flow Analysis

The goal of our analysis is to determine whether it is possible for tainted data to reach sensitive sinks without being properly sanitized. To this end, we have to identify the taint value of variables that are used at these sinks. To achieve 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, 53, 56]. 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 set of 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, 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 thesis.

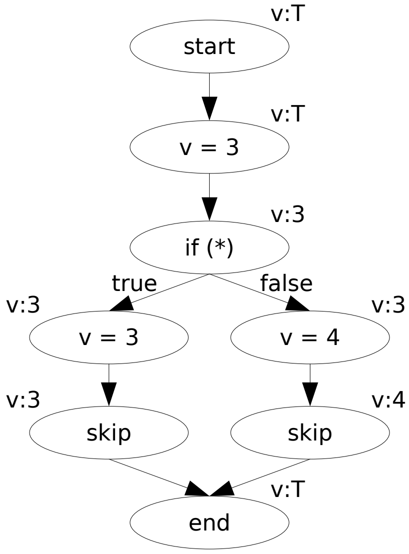

Figure 3.1: Example CFG with associated resulting analysis information.

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 3.1 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.

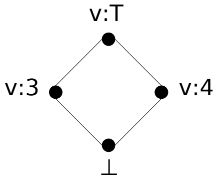

Figure 3.2: A simple lattice.

An important concept used in the theory of data flow analysis is that of a *lattice*[1], 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 Figure 3.2 (as Hasse diagram [84]). Note that an additional *bottom element* $\perp$ is required by the analysis algorithm for marking nodes as "not visited yet" at the beginning (to prevent premature termination of the algorithm). Each line in Figure 3.2 indicates an ordering between the elements at its end points with regard to precision. For instance, the element (`v`:3) is more precise (less conservative) than (`v`:$\top$), written as (`v`:3) $\sqsubset$ (`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, (`v`:3) $\sqcup$ (`v`:4) = (`v`:$\top$), and (`v`:3) $\sqcup$ ($\perp$) = (`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 Figure 3.1. A more comprehensive introduction to lattice theory can be found in Appendix A.

In this thesis, the notation with regard to lattice structure and the least upper bound operation follows Nielson et al. [56]. Other authors, especially in the advent of data flow analysis (introduced by Kildall [42]), used an alternative, but equivalent notation. In this other notation, more precise elements were located at the top of the lattice, instead of 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 Figure 3.1, 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

[1] 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 [56].

at merge points. As soon as a fixed point is reached (i.e., additional computations do not lead to changes anymore), the algorithm terminates, and the analysis is finished.

3.1.1 Static Analysis Attributes

There exist several attributes of static analyses that determine their precision and efficiency, depending on the presence or absence of these attributes. In general, attributes that increase precision lead to decreased performance. One of the main challenges in static analysis is to achieve a trade-off between precision and performance that yields good results in practice.

Flow-Sensitivity. 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). To see this, consider the following code:

```
$v = 3;       // v:3
$v = 4;       // v:4
$v = 3;       // v:3
```

If this example program is scanned with a flow-sensitive analysis, it associates all three program points with precise information about variable $v (shown in comments next to the statements). A flow-insensitive analysis, however, does not compute information for every single program statement, but for coarser units such as functions, or even the whole program. For the above code, such an analysis would compute only one result for the whole program, namely “v:$\top$”. Obviously, this is a quite imprecise result, as it indicates that “no information is available about $v.” To improve this, it would be possible to adapt the analysis such that it returns the result “v:{3,4}” (i.e., “variable $v can hold one of the values 3 or 4”), which is still less precise than the result of a flow-sensitive analysis. Since a flow-insensitive analysis does not compute information at the granularity of single statements, its results are not affected by reordering program statements within its granularity scope (e.g., within functions). Due to their smaller complexity, flow-insensitive analyses have a better performance than more precise, flow-sensitive analyses.

Interprocedurality. *Interprocedural* analyses take function calls into account, while *intraprocedural* analyses operate only inside a single function. For instance, the following code performs a function call on Line 3, which returns the integer value 4:

```
1 function foo() {
2     $v = 3;           // v:3
3     $v = bar();       // interprocedural ... v:4
4                       // intraprocedural ... v:⊤
5 }
6
7 function bar() {
8     return 4;
9 }
```

If the analysis is intraprocedural, it cannot compute the effects of function invocations. Consequently, in order to compute correct results, it has to make conservative assumptions about the behavior an return values of called functions. For the above example, the effect of the call to "bar" on Line 3 is approximated by considering it to return the unknown value $\top$. In contrast, an interprocedural analysis is able to determine that the return value is 4.

Interprocedural analyses can further be classified depending on whether they are *context-sensitive*, or *context-insensitive*. Context-sensitive analyses distinguish between different call sites to a function, whereas context-insensitive analyses do not make this distinction. Therefore, a context-insensitive analysis confuses the information computed for different calls to the same function. In the example shown in Figure 3.3, context-sensitivity makes it possible to distinguish between the two calls on Lines 3 and 4, and to compute precise information for variables $x and $y. If the analysis is context-insensitive, data flows related to different contexts are merged (by computing the least upper bound), leading to the return value $\top$ for the two function invocations.

```
1                     //  context-sensitive  |  context-insensitive
2
3 $x = foo(3);        //  x:3                |  x:⊤
4 $y = foo(4);        //  y:4                |  y:⊤
5
6 function foo($p) {
7     return $p;
8 }
```

Figure 3.3: Context-sensitivity.

There exist two different approaches for achieving context-sensitivity, which are described by Sharir and Pnueli in [71]. In principle, both techniques are based on making context information explicit by means of function cloning. For instance, to compute context-sensitive results through function cloning for the example in Figure 3.3, the analysis would take measures that are analogous to transforming the code in the following way:

```
$x = foo1(3);
$y = foo2(4);

function foo1($p) {
    return $p;
}
function foo2($p) {
    return $p;
}
```

After context information has been made explicit by this source-to-source transformation, a context-*in*sensitive analysis can be performed, leading to context-sensitive results. The two approaches presented by Sharir and Pnueli differ in the way how the cloning is performed. In the *functional approach*, a clone is generated for each distinct set of inputs to the function. This is illustrated by the first two columns of the example in Figure 3.4. Since the function invocations on Lines 3 and 5 of this example provide the same input to the function, they can be represented by the same clone (i.e., "foo1"). To express this in more precise terms, the "input" to a function that is used for the cloning decision corresponds to (at least) all data flow information that can affect the behavior (i.e., the return value and side-effects) of the called function. That is, this "input" does not only cover function parameters, but also other influences such as global variables that are used by the called function. An advantage of the cloning policy of the functional approach is that the results for already analyzed functions can be reused at later invocations of the same function. In Figure 3.4, the analysis does not have to analyze the code inside "foo1" for the call on Line 5, as it can reuse the results from the inspection of the previous call on Line 3.

```
1 // Original           Functional           Call-String
2
3 $x = foo(3);          $x = foo1(3);        $x = foo1(3);
4 $y = foo(4);          $y = foo2(4);        $x = foo2(4);
5 $x = foo(3);          $x = foo1(3);        $x = foo3(3);
```

Figure 3.4: Function cloning.

The second approach for achieving context-sensitivity is named *call-string approach.* In its simplest form, this technique creates a function clone for every call to this function. This policy is illustrated by the third column of Figure 3.4, where it generates separate clones for each of the three function calls. However, this can lead to imprecise results in case of call chains, as the following example shows:

```
// Original                // Simple Call-String

$x = foo(3);               $x = foo1(3);
$y = foo(4);               $y = foo2(4);

function foo(p) {          function foo1($p) {
    $z = bar(p);                $z = bar1($p);
    return $z;                  return $z;
}                          }

                           function foo2($p) {
                               $z = bar1($p);
                               return $z;
                           }

function bar($p) {          function bar1($p) {
    return $p;                  return $p;
}                          }
```

In this program, there are two calls to "foo", and one call to "bar". Hence, the analysis creates two clones for "foo", and one clone for "bar". Note that the resulting code still contains two calls to the same function ("bar1"). Consequently, running a context-insensitive analysis on this would again merge the interprocedural information related to this function, and eventually compute the value $\top$ for both $x and $y. To prevent this, it would be necessary to perform an additional cloning step, such that the two invocations to "bar1" are replaced by two invocations to separate clones of "bar1". In this case, a single additional cloning step is sufficient, because the involved call chain had the length 2.

Longer call chains require more cloning steps. The most significant limitation of the call-string approach is that an unlimited number of cloning steps would be necessary for call chains whose length cannot be determined statically (e.g., in the case of recursive calls). Hence, it is necessary to define a number of steps after which the cloning process shall terminate. This number is traditionally denoted as the *k-bound*, and can be used to determine the trade-off between precision and performance for call-string analyses.

In terms of precision, the functional approach is superior to the call-string approach. As far as performance is concerned, a general comparison is difficult, because the actual time spent for the analysis depends on the structure of the program and the k-bound that is used. The functional approach can only by used if the data flow lattice has a finite breadth (if displayed as Hasse diagram), as the analysis would not terminate otherwise. An example for this is shown in the following script:

```
foo(1);
function foo($p) {
    if ($c) {
        foo($p+1);
    }
}
```

Here, function “foo” contains a recursive call that increments its parameter by 1 each time it is called. A functional analysis would create a new clone for every call in this unbounded call chain, and hence, never terminate. This also demonstrates another difference between the two approaches. Functional analysis performs the cloning during the actual data flow analysis, whereas call-string analysis is able to generate the clones in advance, and initiate the data flow analysis afterwards. For more information about context-sensitive analyses, the reader is referred to Florian Martin’s PhD thesis [49].

Path-Sensitivity. The term *path-sensitivity* denotes the ability to recognize impossible paths through the program, and to prevent that these paths reduce the precision of the analysis. For example, consider the following code:

```
1 if ($c) {
2     $x = 3;
3 } else {
4     $x = 4;
5 }
6 if ($c) {
7     echo $x;
8 }
```

For a human, it is obvious that variable $x holds the value 3 on Line 7. Due to the check of the guard variable $c on Lines 1 and 6, it is impossible that the program executes both Lines 4 and 7. A path-sensitive analysis is able to deduce this fact, and uses it to increase the precision of its results.

Soundness / Completeness. 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 report all vulnerabilities that are present in the inspected program. *Completeness* denotes the opposite notion, meaning that a complete analysis must not generate any false alarms (*false positives*). 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 thesis, we follow the interpretation used by Johnson and Wagner in [34], and by Shankar et al. in [70], which corresponds to the explanations given above.

Attributes of our system. Our system achieves a high level of precision by being flow-sensitive, interprocedural, and context-sensitive. Precision could be further enhanced by equipping our scanner with path-sensitivity. However, during our empirical evaluation, we found that the current configuration performs well in practice. Our scanner is unsound with respect to the language features mentioned in Section 3.7.5 (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.

Finally, 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 by Nielson et al. in [56], and a vivid introduction to the topic can be found at [57].

3.2 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 by Aho et al. 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.

3.2.1 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 [61], our tool for generating parse trees for PHP code, is a combination of the Java lexical analyzer JFlex [32], the Java parser Cup [13], and the Flex and Bison specification files from the sources of the PHP interpreter [59]. 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

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.

Table 3.1: Main types of CFG nodes in P-Tac.

to permit the convenient definition of Java actions for constructing the parse tree. The PhpParser package contains a more detailed documentation of these modifications.

3.2.2 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 3.1 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 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.

3.3 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 3.8). These three components (literal analysis, alias analysis, and taint analysis) are 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 [86], 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 address these issues in detail.

3.4 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 literal information are the resolution of variable variables, variable array indices, and variable function calls.

3.4.1 Carrier Lattice Definition

As mentioned in Section 3.1, 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 Figure 3.2, 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 [84]) is infinite as well. Figure 3.5 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 Figure 3.5) 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.

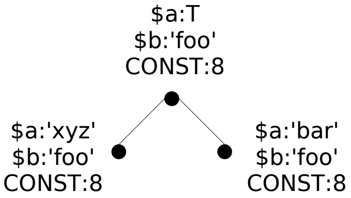

Figure 3.5: Fragment of a literal analysis lattice.

3.4.2 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 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 like "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 step. 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 Figure 3.6. 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

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

Table 3.2: Actions performed by literal analysis and taint analysis for simple assignment nodes depending on the left-hand variable.

really simple. Array elements do not suffer from this uncertainty: While $a[1] surely is an array element, $a is not.

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

Figure 3.6: Arrays can be hidden.

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 3.2, which also contains information about taint analysis and aliases. These topics are dealt with later in this chapter 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 3.7.5). In the rest of this section, we 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 as array[2]. 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 Figure 3.6, the analysis cannot determine that

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

$c[1] is mapped to the literal 7 at the end of the program. In Chapter 4, we describe an improved analysis architecture that allows us to eliminate this source of imprecision.

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, Figure 3.7 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 ⊤ because it is uncertain whether there *really* is no matching node (due to the possibility that the array might contain hidden elements). In Figure 3.6, for example, $c[1] is set to ⊤ 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 actual semantics in PHP). A recursive algorithm for performing all these operations is given in Figure 3.8, where the term *direct element* denotes an array element that is retrieved by adding a single index (the *direct index*) to its enclosing array.

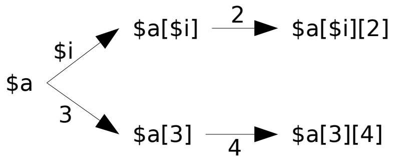

Figure 3.7: Array Tree Example.

To understand cases three and four, one has to consider 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 ⊤. 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

```
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
}
```

Figure 3.8: Strong Overlap Algorithm.

the strong overlap algorithm. Note, however, that taint analysis (discussed in Section 3.7) will perform different operations for cases two and three.

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.

3.4.3 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 3.5 provides a more detailed introduction to PHP references and describes an alias analysis that collects the alias information required by literal analysis. Section 3.6 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 3.8). 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 is described in the next section, resulting in a performance boost without loss of precision. In spite of this fact, we 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 3.7).

3.5 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 runtime. If these variables are aliases only for some program paths, while not for others, they are called *may-aliases*. We 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.

3.5.1 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 3.9 shows a simple example for creating an alias relationship between variables $a and $b (on Line 2). This figure also demonstrates why taint analysis requires access to alias information. Without this information, taint analysis would not be able to decide that the assignment on Line 3 does not only affect $a, but also the aliased variable $b. As a result, we would miss the fact that $b eventually holds a tainted value, which leads to the XSS vulnerability on Line 4. Analogously, the lack of aliasing information can cause false positives.

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

Figure 3.9: Simple aliasing in PHP.

In the past, extensive work has been devoted to the area of alias analysis (e.g., [4, 11, 45, 76, 85], to mention only a few). An overview of existing solutions and open issues is given by Hind in [26]. 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 [4], Steensgaard [76] and

Das [15] 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 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 [59] 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* [59] 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). Furthermore, 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 3.10 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++.

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

Figure 3.10: References in contrast to pointers.

To the best of our knowledge, the issues discussed above have not been addressed in the literature so far. Minamide [50] and Huang et al. [30] briefly mention their use of alias analysis for PHP, but without providing details. Similarly, Liu et al. [47] only briefly mention that they have applied existing pointer analysis algorithms to Python programs.

3.5.2 Intraprocedural Alias Analysis

Figure 3.11 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. Figure 3.12 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 Figure 3.11 would result in empty may-alias information, which deviates from the correct result shown on Line 7.

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

Figure 3.11: Intraprocedural analysis information.

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.

3.5.3 Interprocedural PHP Concepts

Before going into the details of our interprocedural alias analysis, we 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

```
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
}
```

Figure 3.12: Algorithm for the combination operator.

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.

3.5.4 Interprocedural Alias Analysis

The main problem that arises 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 [56] (i.e., it would have an infinite height), the analysis would not terminate in such cases. Our solution to this problem is the following:

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

In the global scope, only global variables are considered. This 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 first give a brief overview of the answers to these questions. A more detailed treatment is 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 3.7.5). In the following sections, we discuss each of the above issues in detail, ordered by increasing complexity of the necessary concepts.

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 [71]. 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 Figure 3.13. 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 10, we simply apply our intraprocedural techniques. As mentioned above, each function only tracks information about global variables and its own local variables. Therefore, the information about the local variables of "a" is removed prior to propagation into "b". The information about global variables, however, is propagated as it is. Inside function "b", the global aliases are modified by the statement on Line 15. On Line 11, 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.

```
skip;  // u{} a{}
a();
skip;  // u{(m.x1, m.x2, m.x3)} a{}

function a() {        // u{} a{}
  $a1 =& $a2;         // u{(a.a1,a.a2)} a{}
  $GLOBALS['x1'] =&
    $GLOBALS['x2'];
  skip;               // u{(a.a1,a.a2) (m.x1, m.x2)} a{}
  b();
  skip;               // u{(a.a1,a.a2) (m.x1, m.x2, m.x3)} a{}
}

function b() {        // u{(m.x1, m.x2)} a{}
  $GLOBALS['x3'] =&
    $GLOBALS['x1'];
  skip;               // u{(m.x1, m.x2, m.x3)} a{}
}
```

Figure 3.13: Aliases between global variables.

Aliases between the Callee's Formal Parameters

Aliases between *formal* parameters appear when there exists an alias relationship between the corresponding *actual* call-by-reference parameters. For instance, function "b" in Figure 3.14 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.

```
a();

function a() {   // u{} a{}
  $a1 =& $a2;    // u{(a.a1, a.a2)} a{}
  b(&$a1, &$a2);
}

function b(&$bp1, &$bp2) {
  skip;          // u{(b.bp1, b.bp2)} a{}
}
```

Figure 3.14: Must-aliases between formal parameters.

For the treatment of may-aliases between formal parameters, additional considerations are necessary. First, recalling that may-alias pairs are unordered, we can identify three types of may-alias pairs that can exist at the time of a function call: (local, local), (global, global), and (local, global). Next, we can distinguish several cases depending on how many elements of a may-alias pair are used as actual call-by-reference parameter (either one or both). Of course, if no element of a may-alias pair is used as parameter, it cannot induce aliases between formal parameters. Table 3.3 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 the case in the second row of

Table 3.3 is shown in Figure 3.15. 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 3.16 shows the exact algorithm that was applied in this case.

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

Table 3.3: May-aliases between formal parameters resulting from calls to a function with signature b(&bp1, &bp2)

```
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 }
```

Figure 3.15: May-aliases between formal parameters.

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).

```
- 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
```

Figure 3.16: Algorithm for adjusting the alias information that is propagated into a callee.

- 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 3.17 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.

```
 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 }
```

Figure 3.17: Must-aliases between formal parameters and global variables.

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 present the notion of *shadow variables.*

Shadow Variables. Our analysis uses two types of special variables for solving the problem mentioned previously. 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 during 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 benefit of shadow variables, consider Figure 3.18, which shows a code snippet covered by Case 1b. At the time of the call to function "b" on Line 8, the local variable $a1 is a must-alias of the global variable $x1. Inside the called function on Line 13, 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 important 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 no longer aliased with \$x1. 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.

```
a();
skip;            // u{(m.x1, m.x2)} a{}

function a() {   // u{(m.x1, a.x1_gs) (m.x2, a.x2_gs)} a{}
  $a1 =&
    $GLOBALS['x1'];
  skip;          // u{(m.x1, a.x1_gs, a.a1) (m.x2, a.x2_gs)} a{}
  b();
  skip;          // u{(m.x1, m.x2, a.x2_gs) (a.a1, a.x1_gs)} a{}
}

function b() {   // u{(m.x1, b.x1_gs) (m.x2, b.x2_gs)} a{}
  $GLOBALS['x1'] =&
    $GLOBALS['x2'];
  skip;          // u{(m.x2, b.x2_gs, m.x1)} a{}
}
```

Figure 3.18: Aliases between local variables and global variables.

The detailed algorithm covering all presented cases can be found in Figure 3.19. The interested reader is referred to Pixy, the open-source implementation of our concepts, which contains 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.

3.5.5 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 [56]. 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 introduction 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

```
- 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
```

Figure 3.19: Algorithm for computing the alias information after a function call.

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 when 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 not frequent in real-world PHP programs.

3.6 Literal Analysis Revisited

Using the information collected by alias analysis, we can extend the basic concepts for literal analysis presented in Section 3.4 to correctly take into account 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 3.4, 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 the last three 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 obtained 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 3.2 (on page 20).

Now that the transfer function for simple assignment nodes is defined, 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, $a is assigned -3. Special care was taken to reflect PHP's implicit type conversion mechanisms (described in the PHP Manual [60]). The treatment of **binary assignment nodes** such as "$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 possess 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 3.5.4, 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, based on the operations in the callee. For global variables, this is straightforward: We simply propagate the values at the end of the callee back to the caller (as done by Sharir and Pnueli in [71]). For local variables, the situation is more complicated. 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 Figure 3.20. 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, obtaining the desired result.

```
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 }
```

Figure 3.20: Modification of caller locals due to a global must-alias.

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 for 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 Figure 3.21, local variables $a1 and $a2 of 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

```
a();

function a() {
  $a1 = 1;            // a1:1
  $a2 =& $a1;         // a1:1, a2:1
  b(&$a1);
  skip;               // a1:7, a2:7
}

function b(&$bp1) {  // bp1:1, bp1_fs:1
  $bp1 = 7;           // bp1:7, bp1_fs:7
  $b2 = 2;            // bp1:7, bp1_fs:7, b2:2
  $bp1 =& $b2;        // bp1:2, bp1_fs:7, b2:2
  $bp1 = 8;           // bp1:8, bp1_fs:7, b2:8
}
```

Figure 3.21: Modification of caller locals due to a formal must-alias.

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 9 of Figure 3.22, 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:⊤ on Line 10.

```
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 }
```

Figure 3.22: Modification of caller locals due to a global may-alias.

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 ignored here, as 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 8 of Figure 3.23, 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.

```
 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 }
```

Figure 3.23: Modification of caller locals due to a formal may-alias.

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.

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.

3.7 Taint Analysis

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.

3.7.1 Carrier Lattice Definition

A variable or constant is said to be tainted if it can hold a malicious value (with regard to XSS) originating from user input that has not been sanitized. 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 $\top$, 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 Figure 3.24. 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 [30, 70].

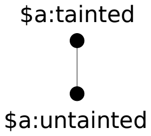

Figure 3.24: Fragment of a taint analysis lattice.

Recall that literal analysis treats non-literal array elements (such as $a[$i]) in a pessimistic way in that they are always mapped to ⊤. 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 Figure 3.25). 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.

```
$a = <user input>;

// $a[1] can be controlled by an attacker

$a = array();

// now $a[1] is no longer controlled by an attacker
```

Figure 3.25: Untainting with `array`.

3.7.2 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 3.2 (on page 20). The “root” field used in Table 3.2 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 3.6, 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 cross-site scripting attacks require to display more data 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, our analyzer processes a specification file on startup which describes the semantics of built-in PHP functions. 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 an untainted value.

3.7.3 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 `print`) is searched for tainted input variables, and a warning message indicating the corresponding line is issued if such a violation is discovered.

3.7.4 Detecting Stored XSS Attacks

Currently, our tool is primarily targeted at the detection of 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 possible that data is not sanitized before it is stored to a database or to the filesystem, 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 taint sources. 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.

3.7.5 Limitations

Currently, our analyzer does not completely support object-oriented features of PHP. It attempts to resolve method calls by means of a simple type analysis, and treats unresolved calls in an optimistic way, meaning that they are considered to return untainted data. Analogously, it is assumed that object member variables are always untainted. 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 the 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 precisely 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.

3.8 Resolving Includes

Virtually all web applications written in scripting languages such as PHP divide their code over several source files. These files are combined at runtime 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 first.

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.

3.9 Empirical Results

We performed a series of experiments with our prototype implementation (written in Java, and called Pixy) to demonstrate its ability to detect previously unknown cross-site scripting vulnerabilities. To this end, Pixy was run on the current versions of six open-source PHP programs. In contrast to C or Java programs, which have one clearly defined entry point where the execution starts (i.e., the main function), web applications written in PHP usually have several different entry points. These entry points correspond to the files visible in the browser's location bar while interacting with the web application. We provided these entry points as input files to Pixy, which automatically resolved further file inclusions. Table 3.4 shows a summary of our results, including the number of entry points and the total lines of code that were analyzed. To determine the line count, we do not factor out files that were analyzed multiple times in different contexts. For example, if an entry file "a.php" includes a file "b.php" twice, the lines of "b.php" are counted twice. Most entry files (together with their transitively included files) were analyzed in less than a minute using a 3.0 GHz Pentium 4 processor with 1GB RAM, even though our prototype still presents many opportunities for performance tuning. There was no analysis run that took longer than five minutes.

In total, we discovered 161 exploitable XSS vulnerabilities in the latest versions of the analyzed programs. In all cases, we informed the authors about the issues and posted security advisories to the BugTraq mailing list [9]. The false positive rate of about 28 % (calculated as $\frac{64}{161+64}$) is relatively low and further alleviated by the fact that many false positives are similar, which makes their recognition easier (see Section 3.9.2 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 3.4. 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.

Program	Entry Files	LOC	T	V	FP	BugTraq ID
DCP Portal 6.1.1	22	61,617	6.0	61	15	427,175
MyBloggie 2.1.3beta	6	20,326	58.3	13	5	427,182
Open Searchable Image Catalogue 0.7	14	14,281	1.5	6	0	435,380
QaTraq 6.5	146	4,275,234	10.7	48	14	438,151
Qdig 1.2.9.2	4	5,385	54.5	2	3	18,653
TxtForum 1.0.4-dev	15	4,398	1.3	31	17	427,186 427,188
Totals	**206**	**4,381,241**	**11.1**	**161**	**64**	

Table 3.4: Summary of vulnerability reports (T = Time in seconds per file, V = Vulnerabilities, FP = False Positives).

3.9.1 A Case Study: MyBloggie

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

Figure 3.26 shows the code in a simplified and condensed form. The sensitive sink on Line 11 receives a tainted value as input, which is held by the function's second formal parameter ($message). This function is called from Line 8 with $tbstatus as actual parameter. Inside the branches of the preceding if-construct, $tbstatus is either set to the empty string on Line 5 (which is untainted), or is built up from the variable $tbreply on Line 3 (the "." is PHP's string concatenation operator). The value of the global variable $tbreply is set by the call to function "multi_tb" on Line 2. A closer look at this function reveals that $tbreply is tainted whenever the first parameter $post_urls of function "multi_tb" is tainted. First, $post_urls is split into an array on Line 16. Afterwards, this array is traversed by the loop starting on Line 17. Inside the loop, $tbreply is assembled from the elements of the array $tb_urls. In effect, since $post_urls can be controlled directly by the attacker (through including the appropriate parameter in a request), this means that the described data flow chain eventually leads to control of the critical $message variable on Line 11.

As already mentioned in Section 3.5.3, 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.

```
if (...) {
  multi_tb($post_urls, ...);
  $tbstatus = $tbstatus . $tbreply;
} else {
  $tbstatus ="";
}

message(..., $tbstatus);

function message(..., $message ) {
  echo $message;
}

function multi_tb($post_urls, ...) {
  global $tbreply;
  $tb_urls = split('( )+', $post_urls, 10);
  foreach($tb_urls as $tb_url) {
    $tbreply .= $tb_url;
  }
}
```

Figure 3.26: Vulnerability in MyBloggie (simplified).

3.9.2 False Positives

A majority of the reported false positives (41 of 64) were due to "impossible" program paths. Figure 3.27 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 }
```

Figure 3.27: False positive due to impossible path (simplified).

The second largest group consisting of 14 false positives was due to more or less complex `if` constructs that are responsible for untainting a critical variable.

From a syntactic point of view, since these constructs had no `else` branch, it might be possible that none of the branches is taken, leaving the variable tainted. However, although syntactically incomplete, the constructs appeared to be semantically complete, as we did not find a way to induce a bypassing condition.

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

As described by Sharir and Pnueli in [71], there are two types of context-sensitive interprocedural analyses, namely *call-string* analysis and *functional* analysis. Functional analysis usually provides more precise results than call-string analysis. In general, none of the two analyses is faster than the other *per se*, since their performance largely depends on the call graph of the analyzed program. However, for one entry point to MyBloggie, it turned out that functional analysis created a large number of contexts during the interprocedural analysis. To address this problem, we performed taint analysis as instance of a call-string analysis with one-element call-strings for scanning MyBloggie (while for the other programs, the functional approach worked fine), which resulted in two additional false positives. We believe that the imposed performance penalty of performing a functional analysis can be effectively reduced by refining some of the internal mechanisms of our analysis (such as the workset order, or the merging of equivalent contexts into one).

The remaining two false positives resulted from validation using regular expressions, which is currently not supported by our prototype. Figure 3.28 shows an example from TxtForum. On Line 1, the programmer validates the variable $_POST[reg_username] by checking whether it contains three to 20 word characters (i.e., letters, digits, or underscores) and hyphens. If the variable does not conform to this specification, the program exits on Line 3. As a result, the `echo` statement on Line 6 is harmless, as control flow reaches this point only if the variable has passed the validation check.

```
1 if (preg_match('#^[\w-]{3,20}$#', $_POST[reg_username]) == 0) {
2   echo "Invalid username!";
3   die();
4 }
5 ...
6 echo "User <b>" . $_POST[reg_username] . "</b> created!";
```

Figure 3.28: False positive due to regular expression validation.

3.9.3 File Inclusion Effectiveness

Table 3.5 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, and each entry was processed in less than 15 seconds. The third and fourth columns show the average number of literal and non-literal includes that were resolved per file. This demonstrates that non-literal includes generally occur more frequently than literal includes, and, as a result, the need for an intelligent resolution algorithm that is able to handle non-literal cases. Otherwise, a significant number of inclusions would be missed, leading to both false positives and false negatives.

Program	**Iterations**	**Resolved Includes**		**Unresolved Includes**
		literal	**non-literal**	
DCP Portal 6.1.1	3.9	0.9	5.8	1.9
MyBloggie 2.1.3beta	3	6.5	12.3	0
TxtForum 1.0.4-dev	1.5	1.8	2.6	1.7
Open Searchable Image Catalogue 0.7	1.0	2.8	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

Table 3.5: Summary of file inclusions (average numbers).

All non-literal includes that could not be resolved assemble the names of the files to be included from dynamic input (mostly from user input, such as cookie fields and POST values, and sometimes from file contents). A close manual inspection of such cases is advisable, since they represent potential security leaks. If an attacker has control over the names of the files that are to be included, it might be possible to inject arbitrary scripts (i.e., arbitrary PHP code) into the program. Most of the cases we encountered are harmless and similar in structure to the first inclusion shown in Figure 3.29. In this example, it is impossible to include a remote file (e.g., located on the attacker's server) because the name of the included file starts with "lib", and not with a protocol specifier such as "http://". However, it would still permit path traversal attacks through the use of path strings containing elements such as "../..". For instance, an attacker could trick the statement into including the server's "/etc/passwd" file, which would be returned verbatim by PHP. This threat is mitigated by the provided suffix ".inc.php", resulting in the restriction that only files with this extension are included. In one case, however, an include statement such as the second one shown in Figure 3.29 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.

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

Figure 3.29: A harmless and a dangerous unresolvable inclusion.

3.10 Summary

Manual security audits targeted at the detection of web application 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 six 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.

Chapter 4

Taint-Aware String Analysis

In Chapter 3, we have shown that the problem of detecting taint-style web application vulnerabilities in PHP programs can be solved by means of data flow analysis [1, 56]. The presented concepts combined high precision with good performance, and allowed us to detect numerous XSS vulnerabilities in real-world applications using our open-source vulnerability scanner called Pixy. In this chapter, we improve and extend the existing analysis in several points, and achieve the following key contributions:

SQLI Analysis. First, we broaden the coverage of the system by equipping it with an additional SQLI analysis, demonstrating the feasibility of providing extensions for previously undetected classes of vulnerabilities. The precision of the SQLI analysis has been enhanced such that it is able to detect a subtle, but dangerous type of vulnerability. For this type of vulnerability, an attacker is able to launch successful attacks against an application even though its programmer was careful enough to sanitize user-provided values. The reason is that the effectiveness of standard sanitization techniques depends on the exact location of user-provided values within an SQL query. Existing detection techniques are focused on the computation of taint information, which is not sufficient for solving this problem. Instead, it is necessary to augment taint information with information about the structure of the generated SQL queries, and the precise location of user-provided input within these queries. We show how this goal can be achieved by generating *finite state automata that are labeled with taint qualifiers.* This approach is supported by the use of an *extended taint qualifier*, which can take one of three values (in contrast to the traditional, binary taint qualifiers).

Program Capability Analyses. We also perform two program capability analyses that provide information about how an application interacts with a back-end database or the filesystem. Database capability analysis reuses the information about the structure of SQL queries computed by SQLI analysis. By determining the way in which database tables are accessed by an application, it presents interesting opportunities for enhancing the application's resilience against attacks. For instance, this knowledge can be effectively used to restrict an application's database access privileges to a required minimum. As a consequence, even when an attacker is able to successfully exploit a novel class of vulnerabilities, the negative effects of this attack attack can be constrained or even eliminated. This is beneficial even for carefully audited applications, as there exists no technique to prove the absence of all vulnerabilities. Likewise, filesystem capability analysis inspects the application's interactions with filesystem objects, and can support the prevention of directory traversal attacks.

Dependence Information. The system described in the previous chapter had a strong emphasis on a specific vulnerability type (XSS), and was characterized by a tight integration of vulnerability-specific parameters into the core analysis. For the detection of XSS vulnerabilities, it performed a data flow analysis that had to be aware of the specific properties of XSS, such as:

- the source of XSS taint values
- the way how these taint values can be propagated or modified throughout the program
- the points where these taint values can lead to vulnerabilities ("sensitive sinks")

This blending of vulnerability attributes with data flow analysis has a number of disadvantages. For instance, if the system shall be used to additionally detect SQLI vulnerabilities, it would be necessary to design and run a completely new data flow analysis, resulting in many redundant computations. Also, even though the definition of SQLI-specific parameters might appear to be straightforward from a high-level view, integrating these parameters into a data flow analysis requires a lot of knowledge about the internal workings of the system, as well as a considerable amount of work. In this chapter, we present a solution to this problem by decoupling data flow analysis from the actual security analysis. Now, data flow analysis is only responsible for calculating dependence information and for providing *dependence graphs* to subsequent security analyses. This way, it is possible to specify the data flow analysis in a completely generic way, such that it is unaware of the type of vulnerability that will eventually be searched for. A key advantage of this approach is that the generated dependence graphs correspond to *taint flow traces*, which provide valuable insights about *why* a security analysis reports a vulnerability. They show, in an intuitive way, where the tainted value initially originates from (e.g., a value submitted through an HTML form), and how this value eventually reaches a sensitive sink (e.g., across assignments and function calls). This considerably speeds up the necessary manual inspection of vulnerability reports by a human operator. Moreover, dependence graphs permit a higher precision in detecting flows of taint values between array variables. Another advantage is that by decoupling the security analysis from the data flow analysis engine, the specifications for security analysis become smaller, easier to understand, and easier to maintain. All that the security analysis needs is a dependence graph provided by the data flow analysis. No knowledge about the inner workings of data flow analysis is required. Also, data flow analysis can be performed once, and reused for an arbitrary number of client security analyses. In addition to SQLI and capability analysis, we show how the XSS analysis from our previous system can be rephrased to operate on the newly generated dependence graphs.

Evaluation. To test our concepts in practice, we have integrated them into Pixy, our web application vulnerability scanner. We performed a comprehensive evaluation of four security analyses (SQLI, XSS, database capability, and filesystem capability analysis) by analyzing seven real-world PHP applications, improving the coverage and quality of recent results in the literature. During our experiments, we have detected 197 SQLI and 172 XSS vulnerabilities with a low false positive rate. The capability analyses have extracted the desired information with a high precision, and provide useful insights into the scanned applications.

4.1 Architectural Overview

For achieving the contributions mentioned above, we improve our system's architecture by introducing an additional abstraction layer before the actual security analysis. An overview of our *previous* system's architecture is shown in Figure 4.1. The converter front-end is responsible for transforming a program into a set of control flow graphs (CFGs) suitable for data flow analysis. For this, it requires the name of an entry file, which corresponds to the file that would be displayed in the browser's location bar during a normal web session. Files included by this entry point (using statements such as `include` and `require`) are parsed and transformed recursively. After a CFG representation has been generated, it is passed to an alias analysis for extracting supplementary alias information. Finally, an analysis specialized on tracking the flow of XSS taint qualifiers is invoked. This XSS analysis tracks the flow of data to determine which variables hold tainted values at which program points. Our XSS analysis is a data flow analysis with several precision-enhancing features. More precisely, it is flow-sensitive, interprocedural, and context-sensitive. In addition, special care was taken to precisely capture data flows that involve the use of array variables (which happens frequently in PHP programs).

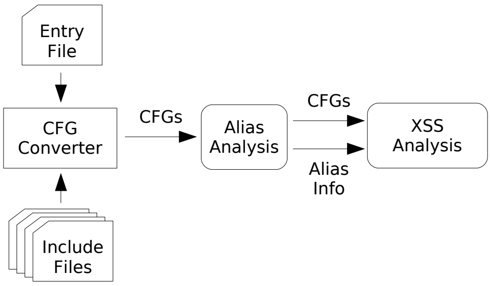

Figure 4.1: Architectural overview of our previous system.

In this chapter, we decouple the computation of data flow from the flow of taint qualifiers, as shown in Figure 4.2. That is, instead of performing an integrated XSS data flow analysis, the results of the alias analysis are first passed to a dependence analysis. This dependence analysis is responsible for

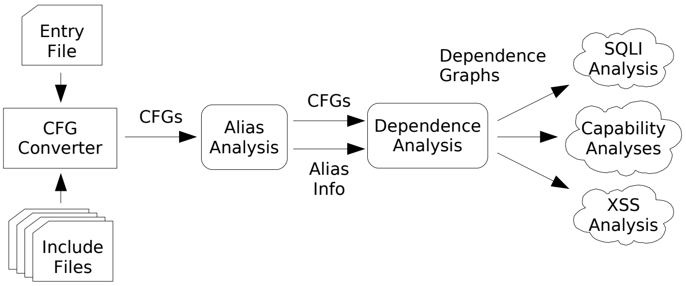

Figure 4.2: Architectural overview of our improved system.

associating every program point with dependence information for each variable. The advantages of a separate dependence analysis phase are the increased ease when developing subsequent security analyses, improved analysis precision, and the fact that dependence graphs help a user to quickly distinguish between false positives and real security vulnerabilities (see Section 4.2). After dependence analysis has finished, it provides subsequent security analyses (which are termed as *client analyses* in the rest of this chapter) with an interface for the retrieval of dependence graphs. These dependence graphs contain all the information necessary for the detection of taint-style vulnerabilities.

4.2 Dependence Analysis

Dependence analysis is a data flow analysis with the purpose of computing, for every variable and every program point, the set of statements upon which the value of this variable may depend. In this sense, it resembles the classic def-use analysis [1]. The information provided by dependence analysis can be used to generate *dependence graphs*, which enable us to bridge the gap between the control flow graph representation of the input program on the one side, and the needs of our security analyses on the other side. Obviously, the notions of data flow analysis and dependence graphs are not new, as they have been used for compiler construction and program optimization for decades. Interestingly, in spite of their maturity and power, Pixy is the only analysis tool that extensively benefits from their advantages for the detection of security problems. These advantages include a reduction of the time necessary for the manual inspection of the generated reports, and an increased precision in the computation of data flows between arrays. Later in this chapter, we discuss how dependence graphs can be used as a building block in a system that detects SQLI and XSS vulnerabilities (Sections 4.3 and 4.5, respectively), and that extracts information about a program's capabilities with respect to database and filesystem accesses (Section 4.4). By extracting data dependence information from a program and representing it in a generic way, subsequent client analyses become smaller, more efficient, and easier to understand.

Figure 4.3 shows a simple code example, together with the information computed by dependence analysis. In the beginning, all variables are uninitialized, which we encode with the symbol `uninit`. Note that the `skip` command represents a no-op statement that does not change data flow information. After

```
1 skip;          // a:uninit  b:uninit
2 $a = 'h';      // a:2       b:uninit
3 $a = $x;       // a:3       b:uninit
4 skip;          // a:3       b:uninit
5 if (..)
6   $a = 'j';    // a:6       b:uninit
7 skip;          // a:3,6     b:uninit
8 $b = $a;       // a:3,6     b:8
9 echo $b;       // a:3,6     b:8
```

Figure 4.3: Dependence analysis example.

analyzing the assignment statement on Line 2, the data flow fact "the value of variable $a depends on the statement on Line 2" is recorded (indicated by the comment on the right side of this line). On Line 3, the value of $a is overwritten, which leads to the new data flow fact "a:3" for this line. On Line 4, there is another `skip` statement, and hence, the data flow information remains unchanged - the value of $a still depends on the assignment on Line 3. Inside the `if`-construct on Line 6, $a is overwritten again. On Line 7, two possible paths through the program merge (the path through the `if`-construct, and the path that bypasses it). Hence, the analysis has to conservatively combine the information from both paths, resulting in "the value of $a may depend on Line 3 or Line 6". Finally, note that on Line 8, the assignment of $a to $b does *not* result in "b:3,6", but simply in "b:8". This is because the alternative would destroy valuable information that is later required for the construction of dependence graphs. Of course, our analysis is also capable of dealing with more sophisticated language features, including arrays, aliases, function calls with parameters, and return values. More specifically, it possesses all precision-enhancing features that were already presented for the previous XSS analysis in Chapter 3.

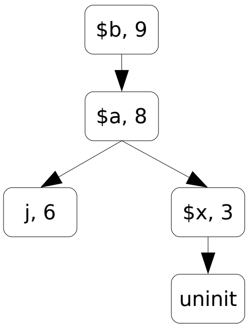

Figure 4.4: Dependence graph for Figure 4.3.

Whenever a client security analysis requests information from the dependence analysis, this information is returned in the form of a dependence graph. This graph abstracts away control flow and only focuses on the data dependencies between variables. For instance, if a client analysis requests information about the data dependencies of $b on Line 9 of Figure 4.3, the dependence graph shown in Figure 4.4 is returned. Its root corresponds to the query issued by the client analysis ("$b on Line 9"). The edge leading to the node labeled with "$a, 8" indicates that the value of $b on Line 9 depends on the value of

$a on Line 8. Since the value of $a on Line 8 conditionally depends on one of two different statements, there are two outgoing edges from this node. The node “j, 6” has no further outgoing edges, as the value of the literal “j” does not depend on anything. The edge from “$x, 3” to the `uninit` node encodes the information that $x is an uninitialized variable on Line 3.

Note that the dependence graph described in the previous example is only one of several dependence graphs that can be extracted from the code in Figure 4.3. The choice of the dependence graph is given by the parameters of the client’s query (i.e., variable and line number). In this sense, there exists an implicit whole-program dependence graph for the complete code in Figure 4.3. Every single dependence graph returned to a client is a slice from this larger graph, and contains only the information that is required by the client.

4.3 SQLI Analysis

Previous approaches to the detection of SQLI vulnerabilities (see Chapter 6) were based on the propagation of binary taint qualifiers (tainted or untainted) through the program. Moreover, they did not attempt to extract and make use of information about the syntactic structure of the generated SQL queries. Unfortunately, these straightforward analyses might miss certain types of SQLI vulnerabilities. To see this, consider the example shown in Figure 4.5.

```
1 $id  = mysql_escape_string($_GET['id']);
2 $pw  = mysql_escape_string($_GET['pw']);
3 $sql = "SELECT * FROM users WHERE id=$id AND pw='$pw'";
4 mysql_query($sql);
```

Figure 4.5: SQLI vulnerability despite sanitization.

On Line 1 of Figure 4.5, a GET parameter is sanitized and copied into variable $id. This variable is used to construct an SQL query on Line 3, which is sent to the database on Line 4. The problem here is that even though the programmer was careful enough to sanitize the user-provided value, this code is still vulnerable to SQL injection. The reason is that $id is not enclosed in quotes when constructing the query string (probably because the programmer expected it to be numeric). The sanitization function `mysql_escape_string` only escapes a small set of special characters, such as quotes and line breaks. As a result, an attacker is still able to alter the syntactic structure of the query [78]. For example, the attacker could provide the value “1 OR 1=2” for the `id` parameter and some arbitrary string (e.g., “abc”) for the `pw` parameter, resulting in the following query:

```
SELECT * FROM users WHERE id=1 OR 1=2 AND pw='abc'
```

This has the effect that the part of the WHERE clause that was initially responsible for checking the user’s password now becomes ineffective due to the injected “OR” keyword. Thus, the authentication mechanism is bypassed, and the attacker can trick the program into retrieving an arbitrary row from the accessed database table (based on the value of `id`).

Systems that are based on simple taint propagation treat the `mysql_escape_string` function as sufficient to sanitize input. Hence, the use of the variable $sql on Line 4 is considered safe. However, SQL vulnerabilities are typically critical enough to allow an attacker to compromise the entire back-end database. For this reason, it would be beneficial to design an SQLI analysis that is sufficiently precise to detect the type of attack described in the previous paragraph. We tackle this problem with two additional mechanisms.

Additional taint qualifier. First, we introduce an additional taint qualifier. In traditional taint propagation engines, a variable can be either tainted or untainted. In our solution, we distinguish between *strongly* tainted, *weakly* tainted, and untainted variables. Strongly tainted values are considered to hold unsanitized values that can be controlled by an attacker. Weakly tainted variables are those that have been sanitized using `mysql_escape_string` or similar functions. To be more precise, a variable is weakly tainted if the sanitization that has been applied to it implicitly assumes that the variable will not be embedded into unquoted regions of an SQL query. All other variables are untainted.

Unfortunately, this more precise distinction between three taint qualifiers is still insufficient for solving our problem. Using the improved mechanism, an SQLI analysis would be able to determine that the variable $sql holds a weakly tainted value on Line 4. However, it would have no information about the *location* of this weakly tainted value inside the constructed SQL query. That is, it could not determine whether the weakly tainted value is inside quotes or not. Thus, it is required to extract more information about the possible structure of the string variable $sql. To this end, we employ an analysis that can approximate the string values that a certain variable might hold at a certain program point, using a finite automaton. Commonly, automata are used as acceptors, that is, they are applied for deciding whether a string belongs to a certain language. For our purposes, we make use of another property of automata (or, equivalently, regular expressions), namely the ability to describe an arbitrary set of strings.

Labeled Automata. Intuitively, a *labeled automaton* such as the one shown in Figure 4.6 would allow us to precisely identify even subtle SQLI vulnerabilities. In this automaton, which represents the possible string values of variable $sql on Line 4 in Figure 4.5, every edge denotes a single-character transition, and the "⟨.⟩" label stands for an arbitrary character (as the exact string values of $id and $pw cannot be determined statically). Solid edges represent untainted values, whereas dashed edges represent weakly tainted values. Strongly tainted values will be represented by dotted edges later in this chapter (the depicted automaton does not contain any edges of this type). It can be seen that this automaton provides information about the points in the query that might contain tainted values. With this knowledge, it is easy to determine that the weakly tainted variable $id (indicated by the first shaded state in Figure 4.6) is not enclosed by quotes and hence, represents an SQLI vulnerability. In contrast, note that the weakly tainted variable $pw is not dangerous because it *is* enclosed by quotes.

To find SQLI vulnerabilities, our main task is to extract labeled automata that describe the structure of the queries that can be sent to the database by the application. More precisely, the following steps have to be performed for each sensitive sink (i.e., for each SQL query):

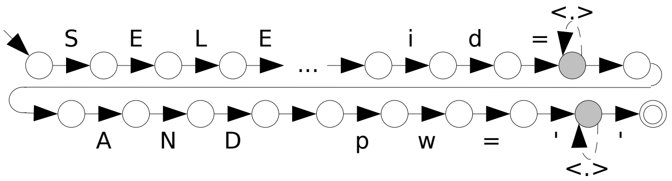

Figure 4.6: Labeled automaton for the query in Figure 4.5.

1. From dependence analysis, retrieve a dependence graph that represents the dependencies of the variable used in the SQL query.

2. Use this dependence graph to compute a labeled automaton for this variable.

3. Decide whether the labeled automaton represents a vulnerability.

Step 1 is the obvious starting point for all security analyses in our system. Without this step, the analysis would not have any information about the scanned program at all. Given a working dependence analysis, this first step is straightforward. Figure 4.7 shows the dependence graph that is returned when the SQLI analysis requests information about variable $sql on Line 4 of Figure 4.5. The graph's root node, "$sql, 5", represents the database query. The edge leaving the root node encodes the fact that variable $sql depends on the concatenation operation on Line 3. The rest of the graph can be explained analogously. Note that the two GET variables ($_GET['id'] and $_GET['pw']) are not explicitly initialized by the program, which is encoded by an edge to a node containing the symbol `uninit`.

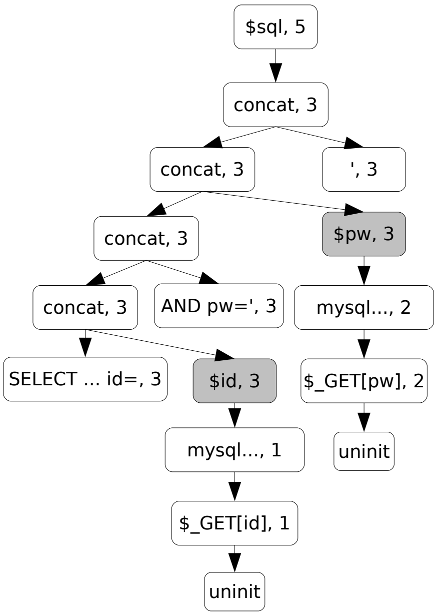

Figure 4.7: Dependence graph for the query in Figure 4.5.

Assume for the moment that the dependence graph does not contain cycles. To compute the automaton, we apply the algorithm shown in Figure 4.8. This algorithm receives the root of the dependence graph as input, and recursively processes all nodes of the graph in a postorder traversal. During this traversal, each processed node is associated (*decorated*) with a separate automaton. Each of these automata describes the possible string values of the corresponding node. For computing such an automaton, the automata of all successor nodes are required as input, which explains the postorder traversal. Once all successors of a node have been successfully decorated, the way how the current node is decorated depends on the type of this node.

```
decorate(Node n) {
  decorate all successors of n;
  if n is a string node:
    decorate n with an automaton describing this string
  else if n is an <uninit> node:
    look at n's predecessor and decorate n accordingly
  else if n is an operation node:
    simulate the operation's semantics
  else if n is an SCC node:
    decorate n with a star automata
    (the taint value of its transition depending
    on the successor nodes)
  else
    decorate n with the union of n's successor automata
}
```

Figure 4.8: Dependence graph decoration algorithm.

In the simplest case, the current node represents a string literal. Such nodes are simply decorated with an automaton that describes exactly this string.

For an `uninit` node, it is necessary to take a look at its predecessor. If the predecessor node represents a variable whose value is taken from user input, such as $_GET[‘a’], the automaton shown in Figure 4.9 is used for decoration. This automaton represents the set of all possible strings (we will refer to such automata as “star automata” from now on). Its sole transition is strongly tainted to reflect the fact that the user input can be malicious. If the predecessor of the `uninit` node is not user-controlled, such as $_SERVER[‘REMOTE_ADDR’], the same automaton is used, but its transition is untainted. Alternatively, it would also be possible to decorate such nodes with a symbolic value instead (e.g., “server_remote_addr”) to retain more of the information from the dependence graph.

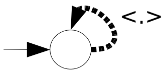

Figure 4.9: Automaton representing a user-controlled value (strongly tainted).

If the node to be processed is an operation node (i.e., a call to a built-in function), then the semantics of this operation has to be simulated. In case of a string concatenation operation, this is simply done by concatenating the automata of the successor nodes. All other operations can be roughly divided into the following categories:

- Weak sanitization functions.
- Strong sanitization functions.
- All other functions (i.e., non-sanitization functions).

Our system is equipped with a list that assigns built-in functions to one of these categories. If a function is not contained in this list, it is conservatively treated as non-sanitization function, such that no vulnerabilities are missed due to missing specifications.

Weak sanitization functions, such as `mysql_escape_string`, are approximated by a star automaton with a weakly tainted transition. For strong sanitization functions, we create a star automaton with an untainted transition. Typical representatives of strong sanitization functions are those that always return a numeric value, such as `intval`. For all other functions (i.e., for all non-sanitization functions), a conservative approximation is to return a star automaton whose transition depends on the taint status of the the function's actual parameters. For example, the function `str_replace($search, $replace, $subject)` replaces every occurrence of the string $search inside the string $subject with the replacement $replace, and returns the result. The taint status of the returned string depends on the parameters $replace and $subject. That is, if either of these parameters is strongly tainted, then the return value is strongly tainted as well. Otherwise, if either of them is weakly tainted, the return value is weakly tainted. Note that the $search parameter does not affect the returned taint value, as it can never appear inside the returned string. To be more precise, the taint status for such functions is the *least upper bound* over the taint status of some of their parameters.

The approximation of most operation nodes by means of star automata may appear overly conservative at first glance. To achieve a higher degree of precision, an alternative would be to model string operation functions with *transducers* (i.e., automata that also produce output). Transducers have already been used for string analysis by Minamide [50], and implicitly by Christensen et al. [12]. In Section 4.6, we will present a transducer-based approach for the recognition of custom sanitization routines. Our empirical results (see Section 4.8) indicate that this partial use of transducers is sufficient for achieving good results in practice.

The final case in the algorithm of Figure 4.8 (Line 13) applies if the current node is a node representing a variable. In a dependence graph, the successor nodes of a variable node represent the values that this variable *may* possess. Different successor nodes correspond to different paths through the program (an example for this was given in our introduction to dependence analysis in Section 4.2). This fact can be translated into an automaton by creating the union of the successor nodes' automata. For example, if a variable $a depends on the two string literals "b" and "c", it means that $a can hold one of these two strings at runtime. An automaton that encodes this information is created by computing the union of the two automata that represent "b" and "c", respectively.

Cyclic Dependence Graphs. The previously described algorithm for transforming dependence graphs into automata is not directly applicable to graphs that contain cycles. For example, the algorithm encounters a problem when a concatenation operation appears inside a cycle. Such a case is shown in Figure 4.10, with the corresponding dependence graph in Figure 4.11. The cycle is caused by the `while` loop on Lines 2 to 4, which contains a concatenation operation that appends a literal to variable $a. In general, the precise modeling of cyclic string operations is a difficult problem [12, 50]. Our solution is to replace strongly connected components (SCCs) in the dependence graph with special SCC nodes, which results in a dependence graph without cycles. This explains Lines 9 to 12 of our decoration algorithm in Figure 4.8. Here, SCC nodes are treated analogously to non-sanitization functions. That is, they are decorated with a star automaton, and the taint value of this automaton's transition is given by the taint values of the SCC node's successors. Our experience and the experimental results presented in Section 4.8 show that this approximation is sufficient for useful results in practice.

```
1 $a = 'h';
2 while ($x) {
3   $a = $a . 'i';
4 }
5 echo $a;
```

Figure 4.10: Cycle in the dependence graph.

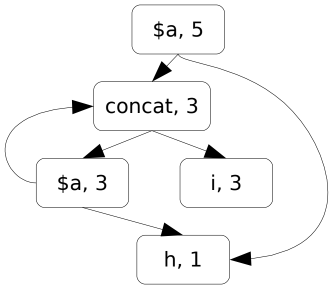

Figure 4.11: Dependence graph for Figure 4.10.

Vulnerability Assessment. When the labeled automaton has been extracted, we analyze this automaton to determine whether there exists a potential SQLI vulnerability. If the automaton contains a strongly tainted transition, there definitely is a vulnerability, independent of the location of these transitions in the automaton. If the automaton contains weakly tainted transitions, we perform a simple data flow analysis on the automaton that determines for each state whether it is "inside quotes" or "outside quotes." If any of the weakly tainted transitions has a source state that is outside quotes, we have successfully detected the subtle kind of vulnerability that has been discussed at the beginning of this section. In the example automaton in Figure 4.6, there is only one state inside quotes (the second shaded node). Hence, the weakly tainted transition that originates from this state is not dangerous in terms of SQLI. However, the other weakly tainted transition in this automaton originates from a state that is not inside quotes (the first shaded node), and thus, indicates an SQLI vulnerability.

4.4 Program Capability Analyses

The SQLI analysis discussed in the previous section offers interesting opportunities for gathering additional information about the behavior of an application, and installing protective measures based on this information. When the SQLI analysis is performed, it creates a list of all SQL queries that the application can send to the database, represented as automata. These queries, or automata, capture the application's *capabilities* with respect to its interaction with the back-end database. Of course, the extracted capabilities are not directly comparable to a simple list of strings. For example, the single automaton in Figure 4.6 (on page 55) encodes an infinitely large set of strings. Based on database capability information, we can determine *how* an application interacts with certain database tables. By associating every table with the set of SQL operations that the application performs on it (e.g., SELECT, or INSERT), the application's developer is presented with useful opportunities.

Leveraging Capability Information. An immediate advantage of the extracted capability information is that the developer is able to compare his *mental* model of how the application interacts with the database with the application's actual capabilities. Conflicts between these two models indicate that there is either a bug in the application, or that the programmer's understanding of the application is faulty. Another advantage of the extracted capabilities is that they can be an aid for writing specification documents. A more immediate benefit in terms of security can be achieved by using the results for enforcing the principle of least privilege. For example, if a table is always accessed by queries that do not change the table's contents, the programmer (or the administrator) can restrict the privileges of the application's database user to read-only access for this table. An even tighter protection can be established by creating database users with different levels of privileges on different tables, and employing these users at the corresponding points in the program. Thus, when an attacker compromises the application, all tables for which the program requires read-only access can be protected from modifications, as well as the tables that cannot be accessed by the compromised database user.

Query Prefixes. To draw a benefit from the extraction of database capabilities, the computed automata have to contain *prefixes* that describe finite strings with a sufficient length. More precisely, the prefixes have to contain the name of the SQL operation (such as SELECT), and the name of the tables that it is applied on. While analyzing the empirical results in Section 4.8, we will refer to such prefixes as “complete”. Shorter prefixes (i.e., those that either lack the table or the operation name) will be considered as “incomplete”.

Filesystem Capabilities. In analogy to the extraction of database capabilities, we can also calculate which filesystem objects are accessed by the application. That is, when analyzing functions that access the filesystem (such as `readfile`), we can extract automata that represent the objects (file or directory names) that these functions operate on. With this knowledge, it is possible to restrict the access rights of the application to only the required parts of the filesystem. Again, we are interested in prefixes of sufficient length. Here, the prefixes can denote either directories or files. In addition, it is also beneficial to take a look at the *suffixes* of the extracted automata, as they might reveal information about the types (extensions) of the accessed files.

Clarifying Remarks. A possible misunderstanding with regard to the use of capability analysis is the following: If the static analysis determines that, for instance, the application can only access files located inside a certain directory, and given that the programmer agrees with this behavior, why should it be necessary to take additional restrictive measures for enforcing this behavior? One answer is that the results of capability analysis do not take into account the possibility of exploiting security vulnerabilities. Consider the code in Figure 4.12. In this example, the programmer intends to read a file inside directory "/xyz", with the file name provided by the client. As expected, filesystem capability analysis extracts the prefix "/xyz/" for this case. However, an attacker is still able to exploit the directory traversal vulnerability contained in the code. For instance, by providing a file name such as "../etc/passwd", he would trick the application into reading the system's password file. Another reason for restrictive measures is that future changes to the code could unintentionally modify the application's capabilities in a harmful way. The damage of such programming bugs would then be limited by the restrictions that were installed previously. And finally, leveraging the extracted capability information for enforcing the principle of least privilege makes the application more secure even under the assumption that it has been carefully audited for vulnerabilities. Currently, there exists no technique that is capable of reliably proving the absence of security problems in an application. Hence, it is essential to create a second line of defense, in case that an attacker discovers a vulnerability that has passed the security audits undetected.

```
$file = $_GET[’file’];
$c = file_get_contents(’/xyz/’ . $file);
```

Figure 4.12: A directory traversal vulnerability.

4.5 XSS Analysis

In this section, we extend our system by an additional analysis for the detection of XSS vulnerabilities. Compared to the XSS analysis applied in the previous version of Pixy, the key advantage of our new XSS analysis is that it provides the user with information about the flows of taint values, which considerably eases the manual inspection of vulnerability reports. Moreover, the additional XSS analysis demonstrates that our improved system can be easily extended to detect new types of taint-style vulnerabilities.

The purpose of XSS analysis is to check every sensitive sink (with regard to XSS) in the program for a potential XSS vulnerability. More precisely, the analysis has to determine whether it is possible that user input reaches a sensitive sink without prior sanitization. In the context of XSS, sensitive sinks are those functions that return data to the user (such as `print` or `echo`). To achieve this goal, the analysis repeats the following steps for each of the sensitive sinks present in the program:

1. Query dependence analysis for an appropriate dependence graph for the current sensitive sink.

2. Transform this graph into an XSS taint dependence graph, taking into account the semantics of built-in PHP functions with regard to the propagation of taint values.

3. Locate tainted variables in the resulting graph. If there are such variables, a potential XSS vulnerability has been discovered.

Step 1 is identical to the first step applied by SQLI analysis (Section 4.3). In Step 2, the dependence graph has to be transformed into an XSS taint dependence graph. This is necessary because a data dependence between two variables does not always imply that both variables possess the same taint status. That is, one variable $x can be tainted, while another variable $y can be untainted, even though there is a data dependence between $x and $y. This fact is due to sanitization routines, which can interrupt the flow of the taint status from one variable to the other. For instance, consider the example shown in Figure 4.13, with its dependence graph in Figure 4.14. Even though the value of $a depends on the value of $_GET[‘a’], the taint qualifier of the GET parameter does not flow to $a because the XSS sanitization function `htmlentities` is invoked on Line 3. For the dependence graph in Figure 4.14, the XSS taint dependence graph that results from it (shown in Figure 4.15) is identical except for the nodes that are enclosed in a dashed frame. In the XSS taint dependence graph, these nodes are replaced by a special “sanitization” node that indicates the use of a sanitization function.

```
1 $a = $_GET[’a’];
2 $b = $_GET[’b’];
3 $a = htmlentities($a);
4 $out = $_SERVER[’REMOTE_ADDR’] . ’: ’ . $a . $b;
5 echo $out;
```

Figure 4.13: XSS analysis example.

After the transformation of the original dependence graph into an XSS taint dependence graph, the XSS analysis reaches Step 3. In this step, the analysis checks whether there are any remaining tainted variables in the graph. To this end, it takes a closer look at all those variables that have not been *explicitly* initialized by the scanned program (that is, all variables that have an edge leading to an `uninit` node in the XSS taint dependence graph). The reason is that for all variables that have not been initialized by the application, the *implicit* default values set by the PHP environment become effective. For example, the PHP environment implicitly sets the variable $_SERVER[‘REMOTE_ADDR’] to the IP address of the client host before the program is run. If this variable is not explicitly initialized by the program, it retains this implicit default value, which poses no threat with regard to XSS. However, the variable $_GET[‘b’] is set to the value of the user-provided GET parameter `b` by the PHP environment. Hence, this variable is tainted, since it can be given an arbitrary value by an attacker. The mere presence of such a tainted variable in an XSS dependence graph indicates an XSS vulnerability, because it means that the taint status of this variable can eventually reach a sensitive sink.

Compared to the previous XSS analysis of Pixy (see Chapter 3), our analysis has an increased precision with regard to arrays. As we noticed during our experiments, Pixy experienced a loss of precision when analyzing code such as

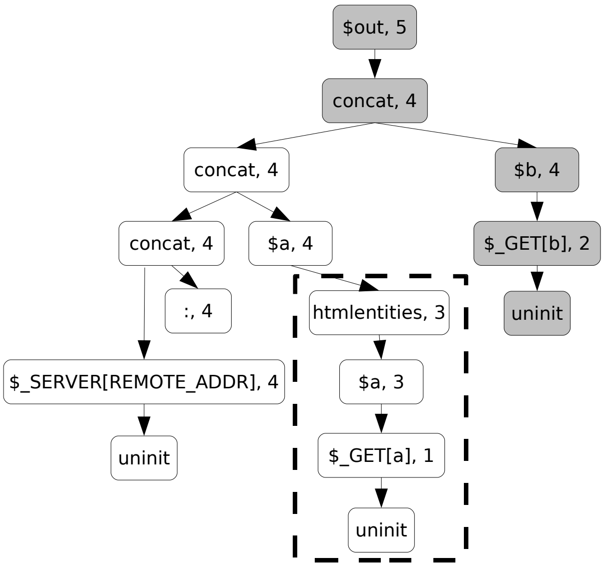

Figure 4.14: Dependence graph for Figure 4.13.

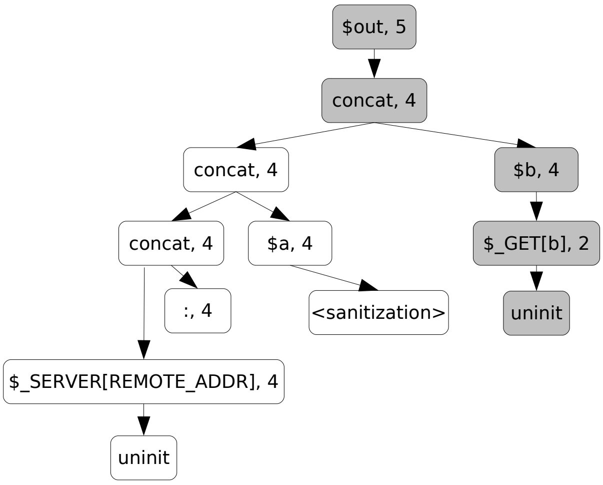

Figure 4.15: XSS taint dependence graph for Figure 4.13.

the example shown in Figure 4.16. On Line 1, $x is assigned to element 1 of array $a. On Line 2, the entire content of $a is copied to $b. A similar copy assignment from $b to $c takes place on Line 3. Finally, on Line 4, element 1 of $c is echoed. Here, it is obvious that $c[1] on Line 4 holds the same value as $x on Line 1. The problem is that $b is never *explicitly* accessed with an array index. Since PHP is dynamically typed, an additional type analysis would be necessary to determine that $b is also an array, and that it contains an element with index 1. Without such a type analysis, information about $a[1] is lost on its way to $c[1], because $b[1] is never explicitly declared in the code. The reason is that it is not possible to simply create a new variable during a running data flow analysis, since this could lead to an infinite loop in the underlying fixed-point algorithm [56]. In our system, this information loss problem can be solved easily during the graph construction phase, without the overhead of an additional type analysis. For instance, imagine that a dependence graph for $c[1] on Line 4 is requested. By consulting the information provided by the dependence analysis (c:3), the graph construction algorithm moves on to Line 3. There, it can deduce that $c[1] depends on $b[1], even if dependence analysis has not provided an explicit entry for $b[1]. This way, graph construction eventually reaches Line 1, and successfully generates the graph shown in Figure 4.16.

```
1 $a[1] = $x;     // a[1]:1
2 $b = $a;        // b:2
3 $c = $b;        // c:3
4 echo $c[1];     // c:3
```

Figure 4.16: Dynamic typing of arrays.

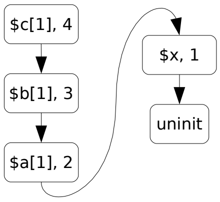

Figure 4.17: Dependence graph for Figure 4.16.

One of the advantages of decoupling the XSS analysis from the underlying dependence analysis is that all XSS-specific parameters can be encapsulated within the XSS analysis. These parameters include sensitive sinks, the semantics of built-in PHP functions with regard to XSS taint flow, and the taint value of uninitialized variables. This encapsulation permits the dependence analysis to be generic, such that it can be reused for other types of security analyses. Another advantage is that developers of additional security analyses do not have to deal with the conceptual complexities of data flow analysis, making these extensions more light-weight and easier to understand.

4.6 Custom Sanitization Awareness

In the previous sections, we have described an analysis that recognizes sanitization that is performed by means of built-in functions (such as `htmlentities`). However, an interesting aspect in the detection of web application vulnerabilities is how *custom sanitization* attempts are handled by the analysis. For instance, consider the example code in Figure 4.18. On Line 1, the value of parameter '`x`', which is provided by a user through a corresponding HTTP GET request, is assigned to the variable $input. On Line 3, this input is sanitized using the standard sanitization function `htmlentities`, and the result is assigned to the variable $standard. On Line 4, the string "Hello" is prepended, and the result is then returned to the user's browser through the `echo` statement on Line 5. On Line 7, an alternative, custom form of sanitization is applied. Here, the character '`<`', which is usually essential to perform XSS attacks, is replaced by the empty string. As a result, the `echo` statement on Line 9 is harmless as well. Finally, on Lines 11 and 12, the input is directly appended to the string "Hello" and returned to the browser. This case constitutes an XSS vulnerability, as an attacker can inject malicious script code via the parameter '`x`', which is echoed without any sanitization.

```
1  $input = $_GET['x'];
2
3  $standard = htmlentities($input);
4  $standard = 'Hello ' . $standard;
5  echo $standard;
6
7  $custom = str_replace('<', '', $input);
8  $custom = 'Hello ' . $custom;
9  echo $custom;
10
11 $missing = 'Hello ' . $input;
12 echo $missing;
```

Figure 4.18: Standard, custom, and missing sanitization.

When analyzing the code in Figure 4.18, our techniques described so far will find the vulnerability that is due to the complete absence of sanitization. Our tool identifies that there is a data flow from the user-supplied parameter ‘x’ to the variable $input (on Line 1) that propagates to the variable $missing (on Line 11), which is eventually output on Line 12. Also, Pixy is equipped with a policy that specifies that all values processed by `htmlentities` can be safely sent back to a user. Thus, the use of variable $standard in Line 5 is correctly flagged as safe.

The situation is more complicated when the function `str_replace` is used for performing custom sanitization. In principle, a static analyzer can choose between two simple strategies when encountering such functions. The first strategy would be to interpret the application of any function that modifies an input (e.g., through string replacement) as an indication that the programmer performed sanitization. If this strategy is used, the analysis would correctly consider the application of the `str_replace` function on Line 7 as a form of sanitization. Of course, this approach suffers from two drawbacks. First of all, the programmer might have simply applied this function to alter the string based on some requirement implied by the program's logic, and changing the string does not imply that the result is safe to be used by a sensitive sink. Second, the programmer could have made a mistake. Even when the input is modified with the intention to make it safe, there is no guarantee that the result is correct, as in the following example:

```
$x = str_replace('<script>', '', $_GET['x']);
echo $x;
```

In this program, the variable $_GET[‘x’] is insufficiently sanitized by deleting all contained `script` tags. This simple sanitization technique can be easily circumvented by embedding JavaScript code in HTML event handlers, which do not require the use of `script` tags. For instance, the event handler `onload` has the effect that the contained JavaScript code is executed as soon as the HTML page is loaded by the user's browser. That is, an attacker could launch an XSS attack by providing the following input:

```
<body onload="alert('XSS')"/>
```

To avoid that such vulnerabilities are missed, Pixy follows the second strategy. That is, it conservatively regards all custom sanitization operations as incorrect.

Even though this approach cannot lead to missed vulnerabilities, it can result in additional false positives whenever a programmer correctly applies custom sanitization to user input.

In this section, we describe an analysis that allows us to assess the effectiveness of custom sanitization attempts. This way, it is possible to eliminate those false positives that were caused by the conservative treatment of custom sanitization mentioned above. To achieve this goal, we present a technique that leverages *transducer-based, implicit taint propagation.*

Basic Automata Computation. Our improved analysis is based on the taint-aware string analysis for SQLI vulnerabilities described in Section 4.3. In this section, our focus is on detecting that the custom sanitization on Line 7 of Figure 4.18 is correct. Since this sanitization is performed with regard to XSS analysis, our first step is to adjust the taint-awareness of our string analysis from SQLI to XSS taint values, which is a straightforward engineering task. After this step has been accomplished, our analysis is able to compute the automata shown in Figure 4.19, which represent the static string value "Hello", and the value of the user-controlled variable $_GET['x'].

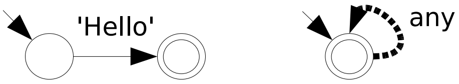

Figure 4.19: Automata for the string "Hello", and for an unknown, tainted string.

Precise Function Modeling. For the analysis of custom sanitization, it is necessary to introduce a precise modeling of string-modifying functions (such as `str_replace`) and replacement functions using regular expressions (such as `ereg_replace` and `preg_replace`). A suitable algorithm was presented in the natural language processing community by Mohri and Sproat [52]. This algorithm is based on the use of *finite state transducers.* A transducer is an automaton whose transitions are associated with output symbols. This way, it is not only able to accept (or reject) input strings, but it also produces output for each input string. By replacing the conservative and imprecise modeling of `str_replace` with Mohri and Sproat's algorithm, the effect of applying string-modifying operations on a set of strings (represented by an automaton) can be captured.

For example, when using Mohri and Sproat's algorithm to analyze the string operations on Lines 7 and 8 in Figure 4.18, we receive the automaton shown in Figure 4.20. This automaton precisely captures the possible values of the variable $custom. That is, it describes the set of strings that start with the prefix "Hello", and end with a suffix that does not contain the '<' character. Unfortunately, the computed automaton does not distinguish between tainted and untainted transitions anymore. Instead, it simply assumes all transitions to be untainted. This is because Mohri and Sproat's algorithm is not designed to work on taint-aware automata. We will present a solution to this problem later in this section.

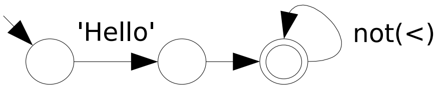

Figure 4.20: Automaton after the replacement of "<".

Vulnerability Detection Through Intersection. To check whether a program is vulnerable at some sensitive sink, it is necessary to determine whether it is possible that the input to this sensitive sink contains any malicious characters or strings. For instance, an XSS attack requires characters such as '<' to be successful, as they are needed to construct JavaScript or HTML code (even if the JavaScript code is contained in an event handler, as was demonstrated earlier in this section with an example using the `onload` event handler). In our approach, we verify this requirement by intersecting the automaton that represents the sink's input with an automaton that encodes the set of undesired strings (the *target automaton*). If the automaton that results from this intersection is empty[1], it means that none of the undesired strings can be contained in the input, and that this sink is safe.

A simple example automaton that represents a conservative approximation of the undesired strings with respect to XSS is shown in Figure 4.21. This target automaton represents all strings that contain at least one '<' character. Intersecting this automaton with the automaton from Figure 4.20 yields an empty automaton, which means that this input cannot be used to successfully perform an attack. By doing this, we have successfully identified that the applied custom sanitization was indeed effective. In contrast, the intersection of the target automaton with the automaton that represents the potentially dangerous value of variable $_GET['x'] (Figure 4.19) is non-empty, since the unknown value might contain an arbitrary number of '<' characters.

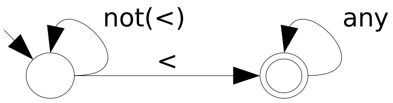

Figure 4.21: Example target automaton for XSS.

Implicit Taint Propagation. Unfortunately, the techniques for function modeling and vulnerability detection described above are still lacking an important ingredient for reaching a sufficient level of precision. The reason is that the algorithm of Mohri and Sproat does not operate on taint-aware automata, but, instead, on traditional automata without taint qualifiers associated to their transitions. That is, the algorithm is not able to propagate taint values through the modeled functions. Without additional measures, this information loss would lead to false positives, as taint information is essential for vulnerability detection. For instance, even the following simple example would cause a false positive:

[1] To be precise: If the resulting automaton accepts only the empty language.

```
$s = "Hello\n";
$x = str_replace("\n", '<br/>', $s);
echo $x;
```

In this code, all occurrences of the '\n' control character are replaced with an HTML line break. Intersecting the automaton that is computed for $x with the XSS target automaton would yield a non-empty result, since $x does contain a '<' character as part of the <br> tag. As a consequence, our analysis would report a vulnerability for this benign example.

A possible approach to solve the problem of propagating taint values through custom sanitization functions would be to modify Mohri and Sproat's algorithm such that it becomes taint-aware. This modification would ensure that the algorithm accepts taint-aware automata as input (i.e., the arguments of the modeled sanitization function), and returns a taint-aware automaton as output.

However, we propose an alternative solution that is efficient, less complex, and less error-prone than a modification of the existing algorithm. Instead of explicitly keeping track of both tainted and untainted values, we concentrate our attention on the tainted parts of the automata. In this *implicit taint propagation*, strings that are statically embedded into the application by the programmer (and hence, untainted) are replaced by the empty string during the automata computation. This has the effect that only tainted strings are explicitly encoded in the automata, and that static, untainted strings cannot lead to false positives any longer.

If used without care, implicit taint propagation can cause false negatives in certain cases (i.e., vulnerabilities might be missed). Consider the following (rather contrived) example:

```
$s = 'a';
$x = str_replace('a', $_GET['x'], $s);
echo $x;
```

Here, the program replaces the character 'a' inside string $s with a user-provided value (from $_GET['x']). If our analysis propagates taint values implicitly (and hence, replaces the value of $s with the empty string), it would incorrectly deduce that the str_replace operation results in the empty string as well. Under the XSS target automaton defined above, an empty string is benign, and, therefore, the vulnerability would be missed. Consequently, it is necessary to compensate the information loss due to the implicit taint propagation with a supplementary "safety net." This additional mechanism corresponds to checking whether the second parameter of str_replace (or, analogously, the replacement parameter of similar functions) is tainted by querying the traditional taint analysis of Pixy. If the parameter is tainted, the result of the function invocation is conservatively approximated with the automaton that describes the set of all possible strings. This ensures that implicit taint propagation does not introduce false negatives into the analysis.

4.7 Implementation

We have integrated our concepts into Pixy, our web application vulnerability scanner. For the construction of labeled automata, we integrated parts of the

BRICS automata library [8]. Custom sanitization awareness was achieved by using the implementation of Mohri and Sproat's algorithm [52] in the Finite State Automata Utilities [20].

4.8 Empirical Results

To evaluate our proposed techniques in practice, we performed comprehensive tests on seven real-world open-source applications (see Table 4.1). More precisely, we ran a full analysis (including SQLI, XSS, database capability, and filesystem capability analyses) on all top-level scripts of the applications in our test set. Note that five of these applications (DCP Portal, MyBloggie, OSIC, Qdig, and TxtForum) were already analyzed for XSS vulnerabilities in Chapter 3. Two applications (Qdig and TxtForum) do not possess a database back-end, which explains their absence in the results for SQL-related analyses. As Table 4.1 shows, the entire test run on every entry file for all applications took about two hours (using a 3.0 GHz Pentium 4 processor with 2GB RAM). Through careful engineering, the performance of our research prototype could be further enhanced. However, even though there are opportunities for improvement, the current system performs well in practice.

Program	**Version**	**LOC**	**Entry Files**	**Time (sec)**
DCP Portal	6.1.1	121,420	22	1,560
MyBloggie	2.1.3beta	9,200	6	3,540
MyEasyMarket	4.1	2,544	22	20
NewsPro	1.1.4	4,251	22	23
OSIC	0.7	4,114	14	1,620
Qdig	1.2.9	5,412	4	480
TxtForum	1.0.4-dev	2,138	15	25
	Totals	**149,079**	**105**	**7,268**

Table 4.1: Applications used for the evaluation.

4.8.1 Results of SQLI Analysis

A summary of our SQLI analysis results is shown in Table 4.2. In total, Pixy detected 197 SQLI vulnerabilities and additionally reported 63 false positives, which results in a false positive rate of 24%.

Figure 4.22 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). A less experienced programmer might be tempted to fix this problem by applying a standard weak sanitization routine, such as `mysql_escape_string`, to the return value of `getVAR`. However, this does not completely remove this vulnerability. The reason is that $id remains weakly tainted on Line 14 and, in addition, is not enclosed by quotes, leading to a subtle SQLI vulnerability as explained in Section 4.3. We simulated the incorrect patch and repeated our analysis for the affected part of the application. The result was that our vulnerability report

Program	**Vulnerabilities**	**False Positives**
DCP Portal	83	14
MyBloggie	31	11
MyEasyMarket	4	3
NewsPro	14	34
OSIC	65	1
Qdig	n/a	n/a
TxtForum	n/a	n/a
Totals	**197**	**63**

Table 4.2: Summary of SQLI vulnerability reports.

still contained this flaw, indicating that the problem of a weakly tainted variable was detected correctly. A simple taint checker that is unaware of the location of a weakly tainted value inside an SQL query would not have recognized this flaw.

```
$file_id = getVAR("id");
$catalogue_id = getValueFromID($file_id);

function getVAR($nm) {
  $tmp = "";
  if (isset($HTTP_GET_VARS[$nm]))
    $tmp = $HTTP_GET_VARS[$nm];
  if (isset($HTTP_POST_VARS[$nm]))
    $tmp = $HTTP_POST_VARS[$nm];
  return $tmp;
}

function getValueFromID($id) {
  $result = mysql_query("
   SELECT * FROM sometable WHERE ID=$id");
}
```

Figure 4.22: SQLI vulnerability from OSIC.

Note that we developed working exploits for every SQLI and XSS vulnerability that we report in this thesis. The reason for accepting this considerable effort is that we believe this approach to be the only reliable method for verifying whether a suspected vulnerability is a *real* vulnerability, and not just another false positive. In several cases during our evaluation, we encountered reports that appeared to be true vulnerabilities at first glance. However, they finally turned out to be false positives when we were trying to construct working exploits.

Under this strict policy, even a case such as the one shown in Figure 4.23 is counted as false positive. The SQL query on Line 3 is a true vulnerability, since $val holds a user-defined, unchecked POST value. The query on Line 5, however, is counted as false positive because we could not think of a way to provide a meaningful exploit that becomes effective on this line, and not already on Line 3. In total, 16 false positives fall into this category.

Three of our tested applications (DCP Portal, MyBloggie, and NewsPro) were also tested by Xie and Aiken for SQL injection in [86]. By comparing our results to their advisory [68], we found that all vulnerabilities that they

```
$val = $_POST["vote"];
$sql = "SELECT * FROM $t_poll_ans WHERE aid = '$val'";
$result = mysql_query($sql);
while($row=mysql_fetch_array($result)) {
  $result1 = mysql_query("
    UPDATE $t_poll_ans SET vote = '$new_val'
    WHERE aid = '$val'");
}
```

Figure 4.23: Being strict about false positives.

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. Since our request for the author's implementation was declined, we were unfortunately not able to perform an in-depth comparison of the two tools.

For MyEasyMarket, our analysis initially reported 34 additional false positives that were due to the use of a regular expressions function for custom sanitization. Our improved sanitization-aware analysis (described in Section 4.6) was able to successfully remove these false positives. Figure 4.24 shows an example for the applied sanitization. On Line 1, all characters except letters, digits, whitespace, and the dot character are removed from the potentially malicious variable $nume. This variable is then passed as parameter to function "logline" on Line 2, which uses the variable to assemble and execute an SQL query on Lines 5 and 6. Due to the regular expressions filter that has been applied on Line 1, an attacker is not able to launch a successful SQLI attack in this case.

```
1 $nume=ereg_replace("[^A-Za-z0-9 .]","",$nume);
2 logline("QUEUE: $nume");
3
4 function logline($line){
5   $sqllog="insert into shopadm values('$line');";
6   mysql_query($sqllog);
7 }
```

Figure 4.24: Regular expressions sanitization from MyEasyMarket (simplified).

Of the total of 63 false positives that we observed, we found that 33 of them (52%) share a common characteristic. Figure 4.25 shows a typical example for this class of false positive. 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 leads to the supposed vulnerability. That is, the code in Figure 4.25 is not vulnerable because the malicious path is a *conditionally* impossible path (i.e., it is impossible given a malicious value for the injected variable). Apart from implicit validation, conditionally impossible program paths can also have other reasons. For instance, Line 4 in Figure 4.26

is secure because it either uses a sanitized value in the query, or an empty value (if $post_id was not set by the attacker). In general, it is a challenging task to eliminate this kind of false positive with static analysis. On the one hand, such an effort bears the risk of severely degrading the scanner's performance, and on the other hand, too aggressive attempts to suppress false warnings could result in missing real vulnerabilities.

```
$lower = chop(strtolower($_POST[reg_username]));
if (file_exists("users/$lower.php")) {
  mysql_query(...$_POST[reg_username]...);
  die();
}
```

Figure 4.25: Impossible paths (1).

```
if (isset($post_id)) {
  $post_id = intval($post_id);
}
mysql_query(...$post_id...);
```

Figure 4.26: Impossible paths (2).

The false positives in the second largest group (27, or 43%) were due to validation using regular expression functions, which is currently not supported by Pixy. 25 of these reports were issued for NewsPro, and an example is given in Figure 4.27. On Line 1, it is checked whether the 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 still considers the variable as potentially dangerous on Line 5, and issues a corresponding alert. This method of validation is recognized by Xie and Aiken's tool by checking the used regular expression against a database of regular expressions that are known to be effective for validation purposes.

In the remaining three cases of false positives, a variable was set dynamically by reading its name from a database table. Since our static analysis is not able to resolve dynamic database content, the value for this variable was set to unknown.

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

Figure 4.27: Regular expression validation from NewsPro.

4.8.2 Results of Database Capability Analysis

As shown in Table 4.3, the extraction of SQL prefixes from the programs that use a database back-end was quite successful. Of a total of 1,238 prefixes, only 14 (1.1%) did not contain the desired information (that is, the SQL operation or the names of the affected database tables were missing). Four of these 14 "false positives" correspond to a programming bug in MyBloggie, where a piece of apparently stale code uses an undefined constant as table name. Two other cases were vulnerabilities, where the table names could not be resolved because an attacker can provide arbitrary values for them. In MyBloggie, four prefixes could not be extracted because this would have required a path-sensitive analysis. Figure 4.28 shows the relevant code. On Line 4 of this code sample, the file viewuser.php is included, but this inclusion only takes place if the variable $mode has been set to a certain value (checked on Lines 1 and 3). Inside viewuser.php, the variable $sql is set to a resolvable value on Line 10, but only if $mode has been set previously. Our analysis cannot determine that this is the case, and conservatively assumes that $sql could also be uninitialized on Line 12. In two cases, a prefix was incomplete because we have not modeled PHP's `foreach` command precisely. A syntactically incomplete, but semantically complete `if` construct was responsible for one false positive. Finally, only in one single case, a loop in the dependence graph led to an incomplete prefix.

Program	**Complete Prefixes**	**Incomplete Prefixes**
DCP Portal	682	2
MyBloggie	160	9
MyEasyMarket	121	0
NewsPro	141	0
OSIC	120	3
Qdig	n/a	n/a
TxtForum	n/a	n/a
Totals	**1,224**	**14**

Table 4.3: Summary of database capability analysis.

```
1  switch ($mode) {
2    ...
3    case "viewuser":
4      include viewuser.php;
5    ...
6  }
7
8  /* in viewuser.php */
9  if (isset($mode)) {
10    $sql = ...;
11 }
12 mysql_query($sql);
```

Figure 4.28: Incomplete prefix in MyBloggie.

As an example for how the extracted database capability information can be used in a security context, we were able to deduce that NewsPro uses eight database tables. Two of these tables are accessed read-only during normal operation (they were filled with appropriate values during the installation of the application). This knowledge can immediately be taken advantage of by restricting the rights of the application's database user accordingly. If, at some point in the future, an SQLI vulnerability is successfully exploited, the attacker will not be able to corrupt these two tables. Another table is accessed only

	Prefixes		**Suffixes**	
Program	**Complete**	**Incomplete**	**Complete**	**Incomplete**
DCP Portal	85	0	85	0
MyBloggie	3	1	3	1
MyEasyMarket	8	2	4	6
NewsPro	49	8	56	1
OSIC	16	14	16	14
Qdig	0	4	2	2
TxtForum	79	0	79	0
Totals	**240**	**29**	**245**	**24**

Table 4.4: Summary of filesystem capability analysis.

via SELECT and UPDATE queries, meaning that the number of rows in this table never changes. This invariant can also be enforced to prevent the deletion or addition of rows. The most comprehensive way of utilizing the extracted prefix information would be to translate it into an access pattern that rigorously adheres to the principle of least privilege. For every single access to the database, the extracted prefix provides enough information to generate an appropriate database user with minimal rights. By retrofitting the application with these database users, the damage of future SQL injection attacks can be minimized.

4.8.3 Results of Filesystem Capability Analysis

Table 4.4 lists the results that we obtained from our filesystem capability analysis. Here, the rate of incomplete extraction is higher than for the database capability analysis. 29 (10.7%) of the prefixes and 24 (8.9%) of the suffixes could not be extracted. The majority of these incomplete extractions originates from OSIC (14 prefixes, 14 suffixes) and NewsPro (8 prefixes, 1 suffix). These are caused by the loss of information due to the fact that the programs store values in a database and retrieve them at a later stage. For instance, OSIC stores the name of a certain directory in a database table. Later, this name is read from the database to access files within that directory.

By inspecting the extracted prefixes of DCP Portal, it turned out that all filesystem operations either access objects inside the application's "themes" directory, or a specific HTML file in the "modules" directory. Therefore, it would be possible to restrict the permissions of this web application to these objects. This way, the potential damage that an attacker can do is limited, even if he managed to exploit a command injection or directory traversal vulnerability. A further look at the extracted suffixes shows that all accessed files have the extension ".htm", meaning that all accesses to files with other extensions should be disallowed, or at least, reported as an indication of a security breach.

4.8.4 Results of XSS Analysis

Table 4.5 summarizes the results of our XSS analysis. Initially, we had expected to discover fewer SQLI than XSS vulnerabilities, since we assumed that web application developers are more aware of SQLI problems (typically, SQLI vulnerabilities are considered to be more critical). To our surprise, every tested application that uses a database had *more* SQLI than XSS vulnerabilities. In total, Pixy detected 172 XSS vulnerabilities and additionally reported 74 false positives, which amounts to a false positive rate of 30%. Analogously to SQLI analysis, our improved sanitization-aware XSS analysis was able to eliminate 11 false positives that were initially reported due to the use of custom sanitization routines. For the three applications that have been previously examined in Chapter 3, the results have remained identical for the most part, except for the discovery of additional vulnerabilities in DCP Portal and OSIC, an increase in the number of false positives for OSIC and Qdig, and fewer false positives for MyBloggie and TxtForum. The reasons for these changes include the modeling of additional built-in functions, the correction of programming bugs, and the increased precision in array handling due to the use of dependence graphs. For OSIC, we observed a significant increase of both vulnerabilities and false positives. A closer examination revealed that this increase was due to the correction of a subtle programming bug that had previously caused the analysis to silently skip certain parts of the application. This bug was also the reason for the difference of eleven false positive reports for Qdig. The generated dependence graphs turned out to be highly useful during the inspection of Pixy's reports. In particular, they allowed us to conveniently trace back the causes of taint values to their origins, which frequently crossed both function and file borders. This enabled us to quickly distinguish false positives from real vulnerabilities.

Program	**Vulnerabilities**	**False Positives**
DCP Portal	76	15
MyBloggie	13	3
MyEasyMarket	4	2
NewsPro	4	14
OSIC	42	12
Qdig	2	14
TxtForum	31	14
Totals	**172**	**74**

Table 4.5: Summary of XSS vulnerability reports.

From the total of 74 false positives, 47 (64%) fall into the "impossible path" class previously discussed in Section 4.8.1. 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 [71], which is used by the dependence 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.

4.9 Summary

In this chapter, we have presented a comprehensive system for detecting a variety of important web application vulnerabilities. In particular, we have developed a novel SQLI analysis capable of detecting even subtle types of vulnerabilities. To improve precision, this analysis combines information about the syntactic structure of SQL queries with extended taint qualifiers, which can take one of three values. Moreover, we leveraged the SQLI string analysis to extract program capabilities with regard to database and filesystem accesses, and have shown how this information can be used in a security context. Finally, we demonstrated that our system can be easily extended for the detection of additional types of taint-style vulnerabilities by equipping it with a new XSS analysis. In our system, both SQLI and XSS analysis provide the user with valuable information about the flow of taint values through the program.

We have integrated the presented concepts into Pixy, our open-source web application vulnerability scanner. Using our implementation, we have performed a comprehensive evaluation on real-world applications, which shows that our concepts are feasible and useful in practice.

Chapter 5

Preventing Cross-Site Request Forgery

In the previous chapters, we presented techniques for the static detection of taint-style vulnerabilities, and evaluated these techniques by scanning real-world applications with Pixy, our prototype implemenation. This chapter is devoted to the mitigation of another threat to web application users. Cross-Site Request Forgery [66, 72, 80] (abbreviated XSRF or CSRF, sometimes also called “Session Riding”) denotes a relatively new class of attack against web application users. By launching a successful XSRF attack against a user, an adversary is able to initiate arbitrary HTTP requests from that user to the vulnerable web application. Thus, if the victim is authenticated, a successful XSRF attack effectively bypasses the underlying authentication mechanism. Depending on the web application, the attacker could, for instance, post messages or send mails in the name of the victim, or even change the victim’s login name and password. XSRF attacks can be quite subtle and not always easy to understand and avoid. Furthermore, the damage caused by such attacks can be severe.

In contrast to the well-known web security problems such as SQL injection and XSS, cross-site request forgery (XSRF) appears to be a problem that is little known by web application developers and the academic community. As a result, only few mitigation solutions exist. Unfortunately, these solutions do not offer complete protection against XSRF, or require significant modifications to each individual web application that should be protected.

In this chapter, we present a solution that provides protection from XSRF attacks. More precisely, our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to users as well as to the web application itself. One important advantage of our solution is that there is only minimal manual effort required to protect existing applications. Our experimental results demonstrate that we can use our prototype to secure a number of popular open-source web applications against XSRF attacks, without negatively affecting the applications’ behavior.

5.1 Cross-Site Request Forgery

In this section, we introduce the concepts and mechanisms behind XSRF attacks in more detail.

5.1.1 User Authentication in Web Applications

HTTP is a stateless protocol that is not able to recognize when a number of requests all belong to a particular user. This is cumbersome when applications have to support user authentication, as there is no straightforward mechanism to identify requests of a user that has already performed a successful login. One way to overcome this problem is to preserve user-specific state in client-side cookies [58]. By inserting a `Set-Cookie` HTTP header into the server's reply, a web application can instruct the client browser to create a cookie with a given name and value (see Figure 5.1). In all subsequent requests to the server, the browser automatically includes this cookie information, using the `Cookie` HTTP header. Based on this cookie information, a web application can then associate requests with certain clients.

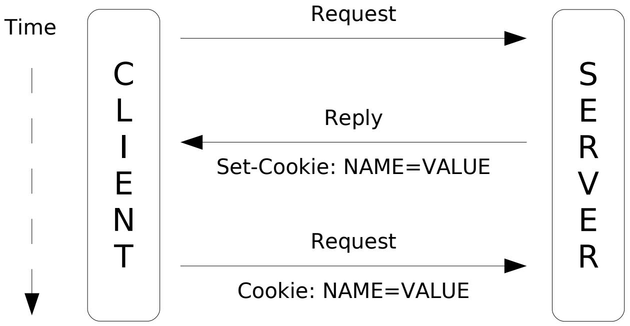

Figure 5.1: Using cookies for client-side state.

Of course, using cookies to store information is only suitable for data that may be freely modified by the client. Because data is stored on the client's machine, it is under the user's direct control. For some web applications, information must not be modified between requests. In other cases, the amount of data that is associated with a certain user is too large to constantly exchange it between the client and the server. To address these issues, web applications typically make use of *sessions*.

A session is established to recognize requests that belong together, and to associate these requests with (session) data stored at the server. To this end, each session is assigned a unique identifier (the session ID), and the client is only provided with this identifier. With each request, the client provides its ID (see Figure 5.2), which the web application can subsequently use to retrieve the appropriate session data.

There are two possibilities to have the client attach the session IDs to each request. The first possibility is to perform *URL rewriting*. In this case, hyperlinks and other request triggers (such as HTML forms) are augmented with an additional parameter that contains the session ID. For instance, the hy-

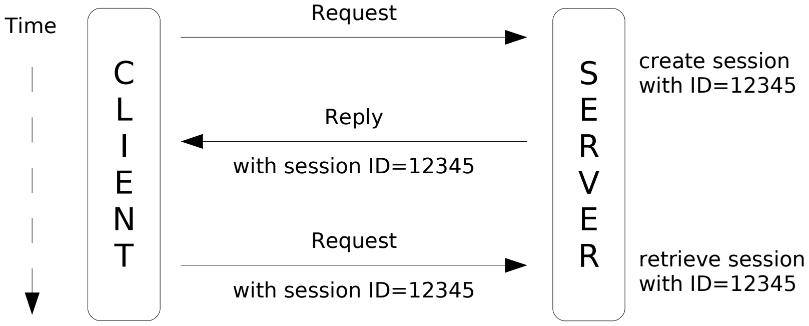

Figure 5.2: Using sessions for server-side state.

perlink with the relative target location `./index.php` might be extended into `./index.php?sessid=12345` to store the session ID with the value “12345”. URL rewriting can be implemented by the application. However, many application runtime and development environments (for example, PHP [59]) already provide an automatic rewriting mechanism to ease the task for web developers. The second possibility to include session information is to set cookies, which are automatically sent by the user browser with each request.

A convenient side-effect of sessions is that they can be used to track the authentication state of a user. For instance, after a successful authentication of the client, a web application could store a boolean value `auth=true` for future reference. When a user continues the current session by sending subsequent requests, the web application can easily determine whether that user is already logged in by consulting this boolean value. As a result, the user is able to perform privileged actions without the need to explicitly submit a password each time. Instead, the authentication occurs implicitly by the underlying session mechanism. That is, the session ID serves as an implicit authentication token.

5.1.2 Exploiting Session Mechanisms

The presented concepts behind web application sessions imply that the session ID temporarily has the same significance as the user’s original credentials. That is, as long as the session has not expired, a web application treats requests with valid session IDs as requests of the user who initially started the session. If an attacker manages to obtain the session ID of an authenticated user, it is possible to issue requests with the same privileges as this user. As a result, the session ID has become a primary target for web application attacks. For instance, one of the goals of cross-site scripting (XSS) attacks is to inject malicious JavaScript code into the reply of a vulnerable application with the aim to leak the session ID to the attacker. In such attacks, the attacker abuses the fact that web applications cannot reliably distinguish between requests with session IDs originating from the legitimate user, and requests with stolen session IDs originating from an attacker.

In contrast, *cross-site request forgery* (XSRF) is a relatively unknown form of attack that is *not* motivated by the attempt to steal the session ID. Instead, XSRF attacks abuse the fact that most web applications cannot distinguish between *intended* user requests, and requests that the user issued because she was tricked to do so. For instance, assume that the online banking application of `www.bigbank.com` receives the following request from an authenticated user:

```
GET /transfer.php?amount=10000&to=7777
```

The application interprets this as a request to transfer 10,000 USD from the user's bank account to the account with number 7,777. Since BigBank's web application does not take into account the possibility of XSRF attacks (unfortunately, this problem is present in many other web applications), it optimistically assumes that the request indeed originated from the HTML form designated for this purpose (shown in Figure 5.3), and faithfully carries out the transaction. In reality, however, the GET request was generated in the following way: After paying an invoice via online banking, the user forgets to log out and proceeds by surfing to some other web sites. One of these sites, `evilxsrf.org`, contains the following hyperlink:

```
<a href='www.bigbank.com/transfer.php?amount=10000?to=7777'>
  Click here
</a>
for something really interesting.
```

As soon as the user clicks on this link, the previously presented GET request is sent to `www.bigbank.com`. Since the user forgot to log out, the session has not been invalidated yet, and the cookie with the session ID still exists. As a result, the user's browser automatically appends the cookie to the request, which is successfully authenticated by the banking application. Without intending so, the user has just transferred a considerable amount of money to some unknown bank account.

```
<form action="transfer.php" method="get">
  To: <input type="text" name="to"/>
  Amount: <input type="text" name="amount"/>
  <input type="submit" value="Submit"/>
</form>
```

Figure 5.3: Legitimate money transaction form of `www.bigbank.com`.

The described, simple attack will probably work only against users that are not security-aware and have limited knowledge about the mechanisms used in web applications. For instance, the value of the `href` attribute will appear in the browser's status bar as soon as the user moves the mouse pointer above the link (although this could be avoided by using JavaScript code that hides the status bar). Also, users become increasingly aware of the security implications of clicking on links in mails. However, the critical request can also be performed through the following `src` attribute of an image tag:

```
<img src='www.bigbank.com/transfer.php?amount=10000?to=7777'>
```

When the user visits the page containing this tag, the browser immediately attempts to retrieve the image by sending the appropriate GET request to `www.bigbank.com`. Compared to the previous case, the user did not even have to actively follow any link, which clearly makes the attack more dangerous. Moreover, XSRF attacks are not limited to GET requests. Figure 5.4 demonstrates how an equivalent POST request can be assembled through an HTML form and automatically submitted by a short piece of JavaScript code. Once again, visiting the malicious HTML page is sufficient for the attack to succeed. Note that although disabling JavaScript would prevent the automatic submission of the form in this case, this measure is not suitable as general cure against XSRF attacks. This underlines the fact that XSRF problems are independent of XSS vulnerabilities and do not rely on the execution or injection of malicious JavaScript code.

```
<form action="http://www.bigbank.com/transfer.php" method="post">
  <input type="hidden" name="to" value="7777"/>
  <input type="hidden" name="amount" value="10000"/>
  <input type="submit"/>
</form>
<script type="text/javascript">
  document.forms[0].submit();
</script>
```

Figure 5.4: Malicious XSRF page for POST parameters.

The analysis of the mechanisms behind XSRF attacks leads to the following observation: As long as a user is logged in to a web application, she is vulnerable. A single mouse click or just browsing a page under the attacker's control can easily lead to unintended requests. Most web applications are not aware of this fact, leaving their users in danger.

5.2 Existing Mitigation Techniques

A common advice for mitigating the XSRF threat that appears frequently in the web development community is to use POST instead of GET parameters. However, as we demonstrated in the previous section, this approach is not adequate for preventing XSRF attacks. It only raises the bar for the attacker, as it closes certain attack vectors such as the use of image tags. In addition, completely removing the use of GET parameters is sometimes not possible when it would result in applications that are more cumbersome for users to navigate and more difficult for developers to implement.

Checking the HTTP `Referer` header would be an effective countermeasure if the web application could rely on its correctness. In the previous example, the request that is generated by clicking the malicious link would contain a referrer to `evilxsrf.org`. By maintaining a white-list of accepted referrers, the banking application could deduce that this request was initiated due to an XSRF attack, and refuse to perform the transaction. Unfortunately, modern browsers can be configured to send empty or even arbitrary values for this header. Moreover, sending the referrer header is discouraged, as it may result in leaking sensitive information to third parties (as mentioned in RFC 2616 [64]). This leads to the question of how to treat *empty* referrer headers. When classifying requests with an empty referrer header as valid, it would become impossible to detect attacks against users who follow the recommendation and disable the transmission of the referrer header. On the other hand, when regarding such requests as XSRF attacks, *all* requests of these users would be rejected. This dilemma is further aggravated by the fact that an attacker can make use of several browser-specific tricks to trigger an XSRF request with an empty referrer [33].

From the previous explanation, it should become clear that XSRF attacks only work when a cookie is used to store the session ID. The reason is that the browser *automatically* includes cookies into requests, even when a user clicks on a simple link. In case of URL rewriting, on the other hand, the session ID has to be embedded into the request trigger (e.g., a hyperlink or a form) explicitly. Thus, when the attacker attempts to create a page with a hyperlink that performs the XSRF request, this link will not contain the proper session ID and thus, will not result in a successful attack. Of course, the adversary cannot prepare the link with a correct session ID, because he has no knowledge about this identifier; otherwise he could use this ID directly to impersonate the authenticated user.

The problem is that cookie-based session management is much more popular and wide-spread for a number of reasons, some of which are even security-related [31, 60, 62]. For example, in URL-based solutions, the session ID appears in the browser's location bar. One implication is that a user might bookmark a page together with the session ID. When visiting the web site via this bookmark, the web server might again associate the session with this ID (this type of session management is called *permissive* and is present, for example, in PHP). As a result, one session ID is used for multiple sessions, increasing the chances for an attacker to successfully steal and exploit the ID. Another possibility is that an attacker could simply peek over a victim's shoulder to steal the session ID (e.g., in a public Internet cafe).

The best solution proposed so far is the use of a *shared secret* (or *token*) between the client and the server to identify the actual origin of a request. For instance, the example banking application from the previous section could be adapted such that the form shown in Figure 5.3 contains an additional, hidden token field. This token must be generated by the application (such that it is not easily guessable by an attacker) and associated with the current session. Requests for financial transactions are only processed if they contain the correct token. The drawback of this approach is the considerable amount of manual work that it involves. Many current web applications have evolved into large and complex systems, and retrofitting them with the mechanisms necessary for token management would require detailed application-specific knowledge and considerable modifications to the application source code. Even more important, there is no guarantee that the modified code is indeed free of XSRF vulnerabilities, as developers tend to make errors and omissions.

XSRF attacks are still relatively unknown to web developers and attackers. Nevertheless, we believe that the attention paid to this class of attacks will reach that of more traditional XSS attacks in the near future as the attack becomes better known and understood. Unfortunately, current mitigation techniques have shortcomings that limit their general applicability. To address this problem, the following section presents a novel and automatic approach for XSRF protection.

5.3 A Proxy-Based Solution

To be useful in practice, a mitigation technique for XSRF attacks has to satisfy two properties. First, it has to be effective in detecting and preventing XSRF attacks with a very low false negative and false positive rate. Second, it should be generic and spare web site administrators and programmers from application-specific modifications. Unfortunately, all existing approaches presented in the previous section fail in at least one of the two aspects.

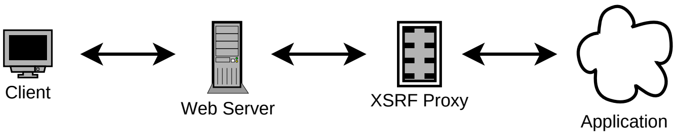

Figure 5.5: Placement of the XSRF proxy.

Our solution to the XSRF problem is to decouple the necessary security mechanisms from the application, and to provide a separate module that can be plugged into existing systems with minimal effort. More precisely, we propose a *proxy* that is placed on the server side between the web server and the target application (see Figure 5.5). This proxy is able to inspect and modify client requests as well as the application's replies (output) to automatically and transparently extend applications with the previously sketched *shared secret* technique. In particular, the proxy has to

- ensure that replies to an authenticated user are modified in such a way that future requests originating from this document (i.e, through hyperlinks and forms) will contain a valid *token*, and
- take countermeasures against requests from authenticated users that do not contain a valid token.

An essential prerequisite for this mechanism is the proxy's ability to associate a user's session with a valid token. To this end, the proxy maintains a *token table* that maps session IDs to tokens (an example is shown in Table 5.1).

Session ID	**Token**
15f34e112fd38c	21647261
a384d70b8b2f0d	91554762
...	...

Table 5.1: Example Token Table.

By decoupling the proxy from the actual application, the XSRF protection can be offered transparently for (virtually) all applications. Note that, alternatively, our proxy could also be located between the client and the web server. However, this case could lead to problems in combination with SSL connections. With our proposed architecture, SSL issues are directly handled by the web server, which eases the tasks that are to be performed by the proxy.

In the following sections, we present a more detailed description of how requests to and replies from the web application are handled, along with illustrative examples.

5.3.1 Request Processing

Figure 5.6 provides an overview of the steps that the proxy has to take during request processing. As a first step, we check whether the request contains a session ID or not. If there is no session ID in the request, it is classified as benign. The reason is that since the request does not refer to an existing, authenticated session, it is not able to perform any privileged actions. Thus, we can safely pass the request to the target application.

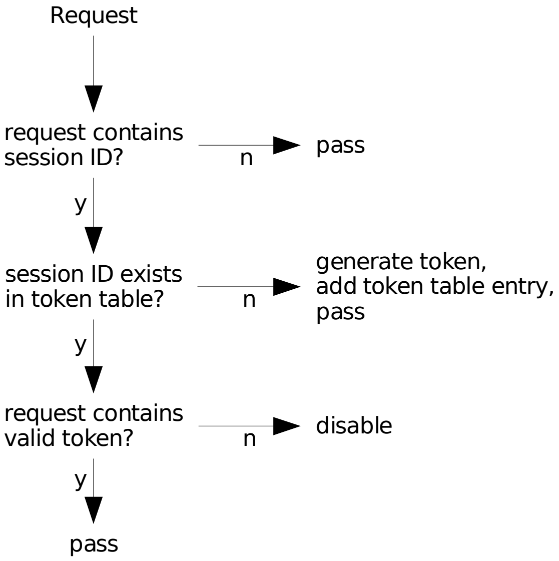
Request
request contains session ID?
n
pass
y
session ID exists in token table?
n
generate token, add token table entry, pass
y
request contains valid token?
n
disable
y
pass

Figure 5.6: Request processing.

If the request does contain a session ID, we consult the token table to check whether there already exists an entry with a corresponding token. If there is such an entry, we require that the request also contains this token. A request that fails to satisfy this condition is classified as an XSRF attack. This is because legitimate requests, originating from a document generated by the protected application, are guaranteed to *always* contain a token when they use a session ID. The reason is that the documents produced by the application are modified such that this token will be present (the exact mechanism to achieve this is described in Section 5.3.2).

The action to be taken when an XSRF request is detected is configurable by the site administrator. In our experiments, we generated a warning message to inform the victim about the attack, together with a (correctly tokenized) link to the application's main page. Note that there is no need to terminate the user's current session when an XSRF attack is detected. After following the link provided in the generated warning message, the user can continue her work normally. An even more convenient, but less educational, alternative would be to instantly redirect the user to the main page, without the need for any additional interaction.

In the case when the request contains a session ID that does not exist in the token table, we have to assume that a new session was established. The proxy generates a new, random token and inserts the token, together with the session ID, into the token table. In addition, the request is passed to the target application.

5.3.2 Reply Processing

As discussed briefly in the previous section, the task of the reply processing step (outlined in Figure 5.7) is to extend the output of a web application such that a subsequent request of the user contains the correct token. This is achieved in a fashion similar to URL rewriting.

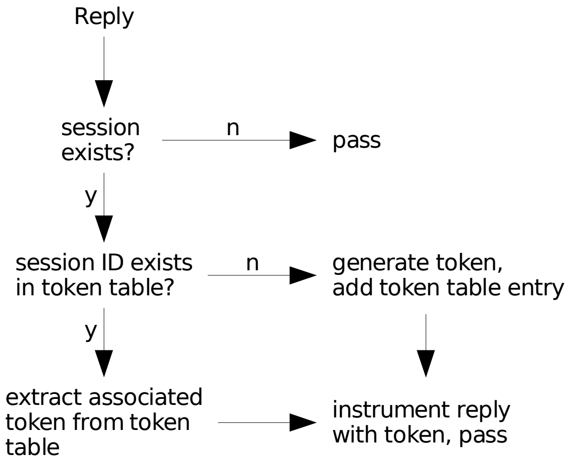
Reply
session exists?
n
pass
y
session ID exists in token table?
n
generate token, add token table entry
y
extract associated token from token table
instrument reply with token, pass

Figure 5.7: Reply processing.

Assume that the proxy has to process an output page of the target application containing the following relative hyperlink:

```
<a href='index.php?action=logout'> LogOut </a>
```

Assume further that the proxy has already determined that the client is authenticated, and that a certain session ID is in use. In this case, it is necessary to rewrite the hyperlink's URL such that it contains the token associated with this session ID:

```
<a href='index.php?action=logout&token=99'> LogOut </a>
```

When the user follows this link, the mechanism has ensured that the proper token is transmitted.

The name of the parameter that stores the token ("token" in this example) can be chosen arbitrarily, but must not interfere with the names of other parameters used by the target application. The token's value ("99") is retrieved from the token table that the proxy maintains.

At this point, an important question is the following: How can the proxy determine whether a client is authenticated or not? For our purposes, we treat the state "a client is authenticated" as equal to "a client has an active session." This is a safe assumption, because XSRF attacks cannot succeed when there is no session information that can be exploited to force the victim into performing privileged actions (that is, actions which require previous authentication) on behalf of the attacker.

The next question is how to determine whether a user has an active session or not. Programming languages such as PHP provide a built-in session infrastructure that could be consulted about whether there exists such a session. However, many applications make use of custom session management techniques. Sometimes, session information is even stored in a back-end database. In such cases, the target application could be instrumented with functions that enable the proxy to issue appropriate queries about the session state. Unfortunately, this would lead to the undesirable necessity to perform application-specific modifications.

We solve the problem of determining whether a session exists in the following way. Basically, there are two cases that have to be distinguished, depending on whether the application sets a session cookie while processing a client's request or not. We can check this by searching the application's reply for an HTTP `Set-Cookie` header. Of course, this approach requires our system to distinguish between session cookies (i.e., cookies that store session information) and cookies that are set for other purposes. While it might be possible to use heuristics to automatically identify session cookies, we currently require the administrator of the system to manually specify their names. Typically, this is straightforward, as many applications make use of the built-in session infrastructure provided by the runtime environment. For example, when PHP is used, the name of the session cookie defaults to "PHPSESSID". If a session cookie *is* set in the application's reply, we assume that there exists a session, and this session has an ID equal to the session cookie's value. If a session cookie is *not* set in the reply, we further investigate the client's request that corresponds to the reply. If this request contains a session ID, we conclude again that there exists a session. Such a situation arises regularly when a client is already logged in, and her browser automatically sends the authentication cookie to the server along with each request.

At this point, note that our approach is safe (i.e., it does not miss any attacks). On the one hand, assume that there exists no session, and that the proxy falsely assumes that there is one. In this case, tokens are included into the applications' documents, but its regular behavior is not affected. On the other hand, if we miss an active session, the reply would not be instrumented with the token. Subsequently, this would lead to a false XSRF alarm for the next user request.

After determining that there exists an active session, we query the token table for an associated token. If there is no such entry, it means that the session has been newly created. Hence, we generate a random token and add a corresponding entry to the token table. Finally, the reply is instrumented with the token before returning it to the client. The following fields have to be modified:

- `href` attributes of `a` tags.

- `action` attributes of `form` tags.
- `src` attributes of `frame` and `iframe` tags.
- `onclick` attributes of `button` tags.
- `refresh` attributes of `button` tags.
- `url` attributes of `refresh` meta tags.

During our experiments, we did not encounter any other fields that required rewriting. However, extending the rewriting engine to take into account more fields would be straightforward.

5.3.3 Token Table Cleanup

The token table should be freed from stale entries regularly to save memory and CPU time. To this end, we extended the table by a third column that holds timestamps, which indicate the point in time when the corresponding entry was last used. When the time that passed since this point is longer than a configurable session life-time (which defaults to 24 minutes, in accordance with PHP's default session life-time), the entry is removed.

5.3.4 Discussion of Attacks against the System

Our proxy cannot prevent XSRF attacks if the target application fails to defend against certain other types of attacks. For instance, if insufficient measures are taken against cross-site scripting (XSS), the adversary could inject malicious JavaScript into the application that steals the user's cookies (containing the session ID). This underlines once more that a reasonable level of security can only be achieved by preparing against a wide range of possible attack vectors.

Our proxy treats all request triggers sent by the target application as legitimate. Hence, all these triggers are automatically instrumented with valid tokens. This implies that if an attacker managed to inject an XSRF hyperlink into the reply of a protected application, our proxy would automatically augment this hyperlink with a valid token as well. This would allow the adversary to induce requests to pages that accept GET parameters. Using POST instead of GET requests would raise the bar for the attacker in this case. However, the attacker could also point the injected link to a site under his control. When the victim's browser requests a page on this site, the `Referer` header could be used to extract the token, allowing the attacker to create valid POST requests as well. To mitigate these problems, the attacker should not be allowed to inject request triggers into an application, a requirement that should be fulfilled by security-aware systems in any case. In addition, clients can decrease their exposure by disallowing the transmission of `Referer` headers (e.g., by configuring their browser appropriately, or by using a firewall).

Given the design of our system, an attacker can obtain the token for any session ID that he presents to the proxy. To this end, the attacker can simply send a request with a particular session ID to the web application and afterwards simulate an XSRF attack against this session. Then, he can extract the associated token from the generated reply (for instance, from the link that is provided along with our warning message). Fortunately, this ability is of no use to the attacker because he has no knowledge of the session ID that a client uses. Otherwise, there would be no need to perform an XSRF attack. Instead, the attacker could directly impersonate the victim using the session ID. Of course, the range of possible session IDs must be large enough to thwart any brute-force guessing attempts.

Finally, the attacker could launch a denial-of-service attack against the proxy. Recall that the token table is extended by a new entry whenever an incoming request contains a session ID that does not exist in the token table. Thus, the adversary could flood the token table with a large number of session IDs with the intention of significantly degrading the proxy's performance. However, the same attack could be launched already against most web servers that track user sessions. The reason is that many applications start to issue session IDs *at the first visit* of a client, and not after the client has logged in. This corresponds exactly to the behavior of our proxy. Hence, for these systems, the possibility for such denial-of-service attacks is not originally introduced by the proxy.

5.3.5 Eliminating State

As mentioned previously, the token table associates session IDs with tokens. This explicit, stateful mapping can be replaced by a mapping that requires no state at all to be stored by the proxy. To this end, a token is computed by applying a hash function based on a secret server key to a session ID. As this approach does not require the explicit storage of session ID to token mappings (the token value can always be computed from the session ID), it eliminates the DoS attack vector sketched in Section 5.3.4. Another major difference compared to the token table approach is that the mappings remain constant over time. When using a token table, mappings change as they are introduced with randomly generated tokens and removed when they expire.

At first glance, static mappings might appear dangerous from a security point of view. An attacker could, in theory, construct an explicit representation of the hash function that is used by sending *all* possible session IDs to the server and writing down the observed tokens in return. By inverting this mapping, the attacker could deduce the session ID from the user's token (modulo hash collisions). This would turn the token into a piece of information worth stealing, increasing the attacker's opportunities. However, reconstructing even small parts of the hash function is not feasible in practice due to the vast number of possible session IDs. In PHP, for example, the space for session IDs comprises 62^{32} entries (32 digits or case-sensitive characters from a to z). For harvesting just 0.01% of all possible mappings in one year, an attacker would have to issue $7 * 10^{45}$ requests per second. An even higher level of security could be achieved by changing the hash function each year, invalidating an attacker's previous harvesting efforts. The resulting convenience penalty for users with active sessions would be minimal, as users with incorrect tokens are immediately provided with a link containing the correct token.

5.3.6 Limitations

The presented concepts rely on the assumption that all request triggers (such as hyperlinks and `action` attributes of forms) are directly available in the output generated by the target application. If this is not the case, reply processing misses certain request triggers, which can result in subsequent false XSRF alarms. For instance, assume a client-side JavaScript is responsible for constructing a hyperlink that points back into the target application. At the time of request processing, this hyperlink is not detected. Hence, the link will eventually lack the necessary token, and result in a false XSRF warning when the client follows it. Fortunately, this is a minor issue for a number of reasons. First, such cases are very rare in practice. During our experiments, we observed this effect in not more than four select boxes. Moreover, adjusting the involved JavaScript such that it also integrates the token into the generated link is a trivial task that does not require any application-specific knowledge. For instance, Figure 5.8 shows such a JavaScript snippet from PhpNuke after our two-line modification (Lines 3 and 4 have been added). The modified code simply extracts the token from the `xsrf_token` attribute of the document's `html` tag and appends it to the constructed URL. The token was previously embedded into the `html` tag by the proxy's rewriting engine. By adapting the JavaScript code at all offending locations, which took less then ten minutes for each application, we successfully eliminated all false warnings. Finally, note that this issue does *not* represent a gap in our security measures. There is no way for the attacker to exploit this issue and launch a successful XSRF attack.

```
1 <select onChange="
2   top.location.href=this.options[this.selectedIndex].value.
3   concat('&xsrf_token='+document.getElementsByTagName('html')[0].
4   getAttribute('xsrf_token'))">
```

Figure 5.8: JavaScript snippet from PhpNuke with modifications.

5.4 Implementation

To demonstrate the feasibility of our concepts, we implemented *NoForge*, a server-side proxy that is able to defend PHP applications against XSRF attacks. As explained previously, this proxy is located between the web server and the protected web applications. To realize this in a straightforward fashion, we decided to implement the proxy as wrapper functions around those PHP applications that we intend to protect. These wrapper functions check the input and output of the application and perform the necessary request and reply processing.

In our test environment, we added simple alias rules to the configuration of the Apache web server that match requests to protected applications and redirect these requests to the proxy wrapper functions. After an incoming request is processed, control is passed to the target application. To be able to process and modify the output generated by the target application before this output is returned to the client, we make use of the output buffering mechanism supplied by PHP. This way, all data generated by the target application is redirected into a buffer that can then be used for the required post-processing step. Note that PHP's output buffers are stackable, which means that this solution works even if the target application performs output buffering itself. During reply processing, the task of instrumenting the output with a token is taken over by a Java program based on the HTMLParser [27] package (a package for parsing and modifying HTML code).

During the implementation of the proxy wrapper routines, we encountered the following problem: The detection of an active session requires that the HTTP headers returned to the client are inspected. Unfortunately, PHP offers no direct mechanisms to access these headers. The output that is written into the proxy's output buffer only contains the message body. We solved this problem by equipping the proxy with additional wrappers around those PHP functions that are responsible for generating the interesting headers (such as `header()`, `setcookie()`, or `session_start()`). Since it is not possible to overwrite built-in PHP functions, we had to write a simple *sed* [69] script that converts calls to these built-in functions into calls to our wrapper functions. For instance, a call to `header()` is automatically rewritten into a call to a wrapper function called `_xsrf_header()`. Note that this trivial modification to the target application is a one-time, fully automatic effort and highly reliable due to its simplicity. Alternatively, the proxy could also be implemented on the network level, where it has full access to the complete HTTP stream.

Another implementation issue we encountered was that if the target application halted program execution using the `exit()` or `die()` function, execution of our proxy stopped as well. As a result, no reply processing was performed. In this case, we successfully applied the same solution as before. By providing additional wrappers around the offending functions, we were able to catch such calls and initiate proper processing of the reply.

To summarize, here are the steps that are necessary for protecting a web application with our prototype implementation:

1. Add an appropriate alias to the Apache configuration.
2. Execute the `sed` script on the target application to enable the proxy's wrapper functions.
3. Specify the cookie names that the target application uses to store session IDs (typically, this defaults to "PHPSESSID").
4. Specify the page that the user shall be redirected to in case of an XSRF attack.

These steps typically take less that five minutes for each application. Clearly, this process is far less time-consuming than manually adapting the whole target application to prevent XSRF attacks.

5.5 Experimental Results

To test our implementation, we chose the current stable releases of the seven largest (in terms of all-time downloads) open-source PHP web applications from SourceForge [74]. The high number of downloads (see Table 5.2) indicates that these applications are popular and wide-spread. Despite the fact that these applications are popular and well-maintained, we quickly discovered XSRF vulnerabilities in five of the applications. The two remaining ones, Gallery 2.0.4 and XOOPS 2.0.13.2, appeared to be immune to XSRF attacks, although we

Application	**Version**	**Downloads**	**Exploits**
phpBB	2.0.19	10,483,075	Delete postings.
			Send postings.
phpMyAdmin	2.8.0.2	9,494,550	Delete databases.
			Create databases.
Gallery	2.0.4	3,937,352	None found.
XOOPS	2.0.13.2	3,448,408	None found.
phpNuke	7.0	2,727,943	Delete messages.
			Add messages.
Coppermine Photo Gallery	1.4.4	1,981,777	Modify user accounts.
			Make existing albums world-writable.
Squirrelmail	1.4.6	1,905,277	Change user information.
			Send mails.

Table 5.2: Tested Applications.

did not conduct exhaustive tests or source code reviews in pursue of XSRF vulnerabilities.

For the five vulnerable applications, we constructed a number of XSRF exploits that modify important data by abusing the privileges of an authenticated user. For instance, we managed to post and delete messages from a forum in the name of the victim, send mail, or even change the user name and password of the current user, resulting in identity theft. A comprehensive list of the created exploits is provided in Table 5.2.

Our first test evaluated the proxy's ability to protect vulnerable applications. After verifying that our exploits were working properly, we installed the proxy and repeated the attacks. This time, every XSRF attempt was correctly detected.

Apart from protecting applications, a central requirement that the proxy has to fulfill is to not interfere with the normal application behavior (both regular behavior as well as behavior in case of errors). To test this property, we can observe and compare the application's behavior without the proxy's protection to the behavior with enabled XSRF protection. If the two results are identical, the proxy succeeded in performing its task transparently. Initially, we considered to perform this test automatically. Unfortunately, we quickly encountered a number of difficulties:

- Forms cannot easily be filled with reasonable input without manual guidance.

- Some syntactic details might change even between semantically equivalent replies (such as date fields or "quotes of the day"), making a straightforward comparison difficult.

- The login procedure and cookies have to be supported.
- Problems with frame or style-sheet display must be detected.

Instead, because of these difficulties, we decided to conduct thorough and systematic manual tests. To this end, we applied the following test procedure for each application:

1. Log in to the application.
2. On the following page, if there is a request trigger (e.g., a hyperlink) that was not activated yet:
   (a) Activate the next unvisited request trigger. If the trigger is a form, test it with correct as well as with incorrect input.
   (b) Hit the browser's "back" button.
   (c) Continue with Step 2.
3. Log out.

Note that these tests also cover the correct behavior of the browser's "back" button, which is a widely used convenience feature that must not be broken by XSRF countermeasures. Also note that the use cases that were targets of our demonstration exploits have already been tested in the previous stage (i.e., while verifying that the XSRF protection works). In addition to this systematic procedure, we also chose some random work flows typical for the application in question. Altogether, we are confident that the coverage of our tests is large enough to give representative results. There was not a single case in which we observed deviant behavior caused by the presence of the proxy.

As far as performance is concerned, we observed no noticeable delay when interacting with the applications protected by our proxy. This is satisfying, as we implemented our prototype without performance in mind, and it still represents many opportunities for optimization. Also, by implementing the proxy and the rewriting engine in a language such as C or C++ instead of in PHP and Java, an additional gain in performance can be expected. Another alternative implementation would be to turn the proxy's application logic into a module for the Apache web server, or integrate it into the already available `mod_security` [51] module.

5.5.1 A Case Study: Sending Mails with SquirrelMail

SquirrelMail [75] is a popular and feature-rich web-mail application written in PHP. Since its initial development release in 1999, the developers fixed several security-critical bugs, among them the XSRF vulnerability described in [54]. The reason for this vulnerability was that the script responsible for sending mails extracted the recipient and the subject from GET parameters. Hence, whenever an authenticated SquirrelMail user at `squirrel-server.com` visited a page containing the following specially crafted image tag, the system generated an email to "x@y.com" with the subject "foo":

```
<img src="http://squirrel-server.com/compose.php?
 send_to=x@y.com&subject=foo&send=1"/>
```

Later versions of SquirrelMail fixed this particular issue by extracting the relevant information only from POST parameters. Although this has raised the bar for an attacker, a successful attack can still be performed with minimal user interaction. For instance, the attacker could trick the victim into visiting a malicious web page, such as the one shown in Figure 5.9. Here, the attacker can provide arbitrary values to the inputs `send_to`, `subject`, and `body`. The short JavaScript code at the bottom of the page has the effect that the form is submitted as soon as the browser has finished loading the page, without leaving the victim any time to react.

```
<form action="http://squirrel-server.com/transfer.php"
  method="post">
  <input type="hidden" name="send_to" value="x@y.com"/>
  <input type="hidden" name="subject" value="foo"/>
  <input type="hidden" name="body" value="bar"/>
  <input type="hidden" name="send" value="Send"/>
  <input type="submit"/>
</form>
<script type="text/javascript">
  document.forms[0].submit();
</script>
```

Figure 5.9: An XSRF exploit page for SquirrelMail (simplified).

When protecting SquirrelMail with our proxy, the previously described attack is detected and successfully mitigated. The proxy checks that the request to the script responsible for sending mails contains a valid token, which is not satisfied by the sketched exploit.

5.6 Summary

In a cross-site request forgery (XSRF) attack, the trust of a web application in its authenticated users is exploited, allowing an attacker to make arbitrary HTTP requests in the victim's name. Unfortunately, current XSRF mitigation techniques have shortcomings that limit their general applicability. To address this problem, this chapter presented a solution that provides a completely automatic protection from XSRF attacks. Our approach is based on a server-side proxy that detects and prevents XSRF attacks in a way that is transparent to users as well as to the web application itself.

We have successfully used our prototype to secure a number of popular open-source web applications that were vulnerable to XSRF. Our experimental results demonstrate that the solution is viable, and that we can secure existing web applications without adversely affecting their behavior.

Currently, XSRF attacks are relatively unknown to both web developers and attackers that are on the hunt for easy targets. However, we expect the attention paid to this class of attacks to soon reach that of more traditional web security problems (such as XSS or SQL injection), and we hope that our solution will prove useful in protecting vulnerable web applications.

Chapter 6

Related Work

6.1 Client-Side Techniques

Client-side techniques in the area of web security are characterized by the fact that they are primarily used by the consumers of web applications, rather than by the providers. In this sense, client-side techniques are either targeted at the protection of their users, or at the detection of security flaws in applications for which no source code is available.

Huang et al. [28] described WAVES, a black-box testing tool for the detection of SQL injection vulnerabilities. Since it does not make use of source code during its analysis, it computes its results by means of fault injection and behavior monitoring. In addition, a machine learning component is applied to guide the testing process. Similarly, a black-box approach is used by Kals et al. [41] in their tool called SecuBat. Apart from SQL injection vulnerabilities, they are also able to detect XSS flaws. One of the main drawbacks of black-box analyses is that they cannot guarantee full coverage of all execution paths inside a program. As a result, parts of the application might remain unscanned, which potentially leads to undetected vulnerabilities.

Noxes by Kirda et al. [43] is a client-side, application-level firewall that offers protection against cross-site scripting attempts. To this end, the authors present techniques for the recognition of outgoing browser connections that might be part of an XSS attack. This recognition process includes the analysis of links embedded in a web page, and the computation of the amount of information that might be leaked to an attacker when a link is followed. The main idea behind this technique is that sensitive information can be transmitted either by a single link that is constructed dynamically inside the user's browser, or by several static links. Based on this information, connection rules are generated on-the-fly, and the user is prompted when a connection violates the existing set of rules.

In an alternative approach to the detection and prevention of XSS attacks on the client side, Vogt et al. [79] enhanced the Firefox web browser with a taint tracking mechanism. This technique dynamically tracks the flow of sensitive information (such as user cookies) inside the browser. Whenever such information is about to be transferred to a third party, an alert is raised. Since it is not possible to detect all types of information flows dynamically [65], an additional static analysis of the web page is applied on-demand.

Johns and Winter [33] presented a proxy-based, client-side solution against XSRF attacks, which is orthogonal to our approach described in Chapter 5. They also build upon the token approach, and additionally propose the use of an outside entity for detecting IP-based authentication. For those cases in which JavaScript code initiates HTTP requests, this code is altered automatically to contain the token. In contrast to this technique, which requires a certain extent of automated program understanding, a manual treatment of these rare cases on the server side appears to provide a more stable and efficient solution.

Apart from the XSRF prevention techniques presented in this thesis, the approach of Johns and Winter outlined above is, to the best of our knowledge, the only scientific contribution to this topic. The general class of XSRF attacks was first introduced by Peter W. in a posting [80] to the BugTraq mailing list, and has since been picked up by web application developers [73]. The mitigation mechanisms for XSRF that were proposed so far by the community (discussed in more detail in Section 5.2) either provide only partial protection (such as replacing GET requests by POST requests, or relying on the information in the Referer header of HTTP requests), or require significant modifications to each individual web application that should be protected (when embedding shared secrets into the application's output). Our solution, on the other hand, attempts to retain the advantage of a solution based on shared secrets, while removing the need to modify application source code. That is, by using a web proxy, we can transparently embed secret tokens into the output of web applications.

Client-side solutions are orthogonal to server-side techniques in terms of capabilities and limitations. On the one hand, client-side tools are preferable for security-aware users who wish to achieve a higher level of protection across a wide range of visited web sites. In the past, several web providers have reacted rather slowly to security threats, leaving their clients in danger. For these cases, an active protection on the client side is able to fill at least a part of the resulting gap. On the other hand, measures taken on the server side are instantly propagated to all clients, such that there is no need for the users to install additional programs for their protection.

6.2 Dynamic Server-Side Techniques

Dynamic techniques that operate on the server side can be divided into two different groups. On the one hand, it is possible to track the flow of tainted values dynamically (i.e., at runtime), and to disrupt program execution whenever a malicious value is used at a sensitive point in the application. On the other hand, approaches based on anomaly detection attempt to recognize suspicious user request before they reach the vulnerable application, and take measures to protect the application from potential attacks.

One of the most prominent examples for dynamic taint tracking is Perl's taint mode [81]. Analogously, Nguyen-Tuong et al. [55] and Pietraszek et al. [63] described modifications to the PHP interpreter that permit the tracking of tainted values. Haldar et al. [23] adapted the Java virtual machine to achieve a similar effect for the execution of Java classfiles, without requiring the corresponding source code.

Su and Wassermann [78] base their work on a formal definition of SQL injection attacks. In their definition, SQL injection occurs when the intended syntactic structure of SQL queries is changed by tainted input. To be able to check whether this policy is violated by a program, they track tainted input dynamically by enclosing it within randomly generated makers. When the program issues an SQL query, the markers indicate the points of the query that contain potentially malicious values.

The server-side techniques outlined above operate by tracking the flow of malicious values. Halfond et al. [25] take the opposite approach with their tool called WASP. In their approach, they identify trusted data sources, and mark data that originates from these sources as benign. Again, the flow of these values through the program is tracked dynamically, and only trusted data is allowed to be used as keywords or operators within issued SQL queries. This corresponds to a white-listing approach (as opposed to black-listing), and has the advantage that sources of untrusted data cannot be omitted accidentally.

Techniques based on the dynamic tracking of taint values have the advantage that the rate of generated false positives is generally low. In contrast to static analyses, they operate on concrete user inputs, and every detected error path corresponds to a true path that the program can take at runtime. The disadvantage of dynamic scanners is that they are mainly suitable for runtime protection of applications, as opposed to pre-release security audits. In the context of pre-release audits, dynamic techniques experience problems regarding the coverage of all possible paths through the program. The number of these paths is generally unbounded, and grows exponentially with each branch in the program. Hence, it is easy to miss vulnerabilities due to program paths that were not taken into account.

As mentioned above, an alternative to tracking taint values through the application is to decide whether user input is malicious before it is passed to the program. For instance, Almgren et al. [2] present an intrusion detection tool that compares the signatures of incoming requests to those of known attacks. The learning process for new attacks is enhanced by keeping track of hosts that launched attacks in the past. In contrast to signature-based systems, systems that apply anomaly detection compute a model of regular, benign requests, and trigger an alert whenever a request does not conform to this model. Kruegel et al. [44] describe an anomaly detection system that is targeted at the detection of web-based attacks. This technique is based on the analysis of the parameters of HTTP queries, which generates parameter profiles for the server-side programs that the queries are directed to. Anagnostakis et al. [3] combine anomaly detection with honeypot technology by introducing the notion of shadow honeypots. To decrease the false positive rate of the anomaly detection component, potentially malicious requests are redirected to an instrumented instance of the protected software, which acts as a honeypot. Using this instance, it is observed whether the request exposes any malicious properties when it is processed. If the request qualifies as benign, the results are passed back to the user, such that the incorrect classification remains unnoticed by the client.

6.3 Static Server-Side Detection of Programming Bugs

This and the following section discuss static server-side techniques, and distinguish between techniques targeted at the detection of classic vulnerabilities or programming bugs, and the detection of typical web application vulnerabilities.

In the past, numerous publications were devoted to the detection of buffer overflows in C programs. For instance, Larochelle and Evans [46] extend the LCLint [19] checking tool to identify buffer overflows. Their analysis is light-weight and efficient, and might miss vulnerabilities. It is annotation-based, relies on the generation and resolution of constraints, and applies a number of heuristics for treating loops in the control flow. CSSV by Dor et al. [16] also requires annotations to perform its analysis. More precisely, it is intraprocedural, such that the user has to provide contracts that specify procedures contained in the program in terms of pre-conditions, post-conditions, and side effects. Ganapathy et al. [22] presented a light-weight static analysis that leverages existing techniques from the linear programming literature to determine the bounds of buffers used in an application.

Engler et al. have published various static analysis approaches to finding programming bugs in C programs. In [17], 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, they present techniques for the automatic extraction of such rules from a given program [18]. Finally, tainting analysis is used to identify vulnerabilities in operating system code where user-supplied integer and pointer values are used without proper checking [5].

CQual [21] by Foster et al. is a tool that allows the introduction of user-defined type qualifiers into C programs. This concept can be leveraged for automatically inferring the taint status (or taint type) of a variable. For instance, this technique was used by by Shankar et al. [70] to detect format string vulnerabilities in C code. Zhang et al. [87] applied CQual to verify the correct placement of LSM (Linux Security Modules) authorization hooks. And finally, Johnson et al. [34] extended CQual with context-sensitivity to detect user/kernel pointer bugs.

6.4 Static Server-Side Detection of Web Vulnerabilities

Huang et al. [30] were the first to address the issue of statically detecting vulnerabilities in the scope of PHP web 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 [29]. A limitation of their work is that they operate on the intraprocedural level, which considerably increases either the number of false positives, or the number of missed vulnerabilities. Besides, essential language elements such as arrays, references, and file inclusions are not supported, and a substantial fraction of PHP files (8% in their experiments) is rejected due to problems with the applied parser.

Livshits and Lam [48] applied an interprocedural taint analysis supported by binary decision diagrams (developed by Whaley and Lam [83]) for finding security vulnerabilities in Java applications. Their analysis is flow-insensitive for the most part, which has the effect that the order of program statements is not taken into account. This means that it is not possible to determine whether a variable was sanitized before or after it enters a vulnerable program point, reducing the precision of the computed results.

Xie and Aiken [86] addressed the problem of statically detecting SQLI vulnerabilities in PHP scripts by means of a three-tier architecture. In this architecture, information is computed bottom-up for the intrablock, intraprocedural, and interprocedural scope. As a result, their analysis is flow-sensitive and interprocedural, and comparable in power to Pixy, the system that we described in Chapters 3 and 4. However, recursive function calls are treated as no-ops, and no alias analysis is performed. Also, they use traditional taint analysis and do not calculate any information about the possible strings that a variable might hold. This makes it impossible to determine *where* in a string tainted information is located. Consequently, certain types of errors (i.e., those subtle SQLI vulnerabilities described in Section 4.3) are missed by their analysis. Our empirical results (Section 4.8.1) show that the vulnerabilities that they discovered are a subset of the vulnerabilities discovered by our system. Hence, together with our XSS and capability analysis, we claim to provide a more precise and comprehensive security analysis. Besides, their vulnerability reports only return a reference to the source of the tainted value. This can considerably slow down the necessary manual inspection process, and make the recognition of complex vulnerabilities difficult. In our technique, the reports are supported by information that allows a reconstruction of the value's path through the program.

String Analysis. Apart from techniques that are focused on the static propagation of taint values, there exist attempts to solve security-related problems in web applications using string analysis. Minamide [50] presented PHPSA, an open-source tool 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 detecting 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.

In [24], Halfond and Orso applied the Java String Analyzer by Christensen et al. [12] to statically extract models of a program's database queries, and used these models as the basis for a runtime monitoring and protection component. 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 statically, which explains the need for an additional runtime component (including all the limitations that come with the use of dynamic analysis).

Wassermann and Su [82] extended PHPSA (mentioned above) to incorporate taint information into the analysis. To this end, they augment context-free grammars with taint labels, and combine this extension with the SQL policy check presented in one of their previous papers [78]. During their analysis, they use three types of taint values (high, medium, none). Compared to our work, the additional taint qualifier has a different purpose, and is used to mark data that is retrieved from the database. Our analysis is more light-weight in that it uses taint-aware automata instead of context-free grammars. Also, the string analysis algorithms proposed by the authors are fairly complex, and less efficient than our approach. The computed grammars are hard to understand compared to our dependence graphs, which inform the user about the flow of taint values in an intuitive way.

To summarize the differences between our static analysis and related work, our approach is aimed at the combination of static taint analysis with string analysis to achieve a high level of precision. While most previous approaches have used either taint analysis or string analysis, or relied on a complex and heavy-weight infrastructure, we present a taint-aware string analysis that embraces the strengths of both (Section 4.3), while retaining good performance and ease of use. In addition, we demonstrate how string information can be leveraged for the extraction of program capabilities, and how the knowledge about these capabilities can be used for enhancing application security (Section 4.4).

Chapter 7

Conclusions

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 approaches for tackling these issues are labor-intensive, costly, and error-prone.

This thesis presented automated techniques in two major areas of web application security. First, we described static analysis techniques for the detection of taint-style vulnerabilities in web applications. In this context, we presented a number of mechanisms for achieving a high level of precision, coverage, and performance. Second, we presented a server-side system for the dynamic recognition and mitigation of cross-site request forgery attacks. This solution can be applied to existing applications easily, and does not interfere with regular program behavior.

To demonstrate the usefulness our concepts in practice, we have conducted empirical evaluations on several real-world applications. The results show that our techniques are feasible, and able to significantly contribute to the security of these programs. Our prototype implementations have been released under an open-source license, and are freely available for download from our homepage.

Bibliography

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

[2] Magnus Almgren, Herve Debar, and Marc Dacier. A Lightweight Tool for Detecting Web Server Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2000.

[3] Kostas Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Usenix Security Symposium*, 2005.

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

[5] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy*, 2002.

[6] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications (Technical Report). To appear, 2007.

[7] Scott Berinato. The Chilling Effect. `http://www.csoonline.com/read/010107/fea_vuln.html`.

[8] BRICS Automata Library. `http://www.brics.dk/automaton`.

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

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

[11] David Chase, Mark Wegman, and F. Ken Zadeck. Analysis of Pointers and Structures. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[12] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *International Static Analysis Symposium (SAS)*, 2003.

[13] CUP: LALR Parser Generator in Java. `http://www2.cs.tum.edu/projects/cup/`.

[14] CVE - Common Vulnerabilities and Exposures. `http://www.cve.mitre.org`.

[15] Manuvir Das. Unification-Based Pointer Analysis with Directional Assignments. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2000.

[16] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[17] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[18] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[19] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *ACM Symposium on Foundations of Software Engineering*, 1994.

[20] Finite State Automata Utilities. `http://www.let.rug.nl/~vannoord/Fsa/`.

[21] Jeffrey S. Foster, Manuel Faehndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[22] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer Overrun Detection Using Linear Programming and Static Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[23] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic Taint Propagation for Java. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[24] William G.J. Halfond and Alessandro Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *International Conference on Automated Software Engineering (ASE)*, 2005.

[25] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Foundations of Software Engineering (FSE)*, 2006.

[26] Michael Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[27] HTMLParser. `http://htmlparser.sourceforge.net/`.

[28] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *International Conference on World Wide Web (WWW)*, 2003.

[29] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Verifying Web Applications Using Bounded Model Checking. In *International Conference on Dependable Systems and Networks (DSN)*, 2004.

[30] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on World Wide Web (WWW)*, 2004.

[31] Java Q & A - Session State in the Client Tier. `http://java.sun.com/blueprints/qanda/client_tier/session_state.html`.

[32] JFlex: The Fast Scanner Generator for Java. `http://jflex.de`.

[33] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In *OWASP Europe Conference*, 2006.

[34] Rob Johnson and David Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Usenix Security Symposium*, 2004.

[35] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2006.

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

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

[38] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[39] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Securing Web Applications Using Taint-Aware String Analysis and Program Capability Analysis (Technical Report). To appear, 2007.

[40] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static Analysis for Detecting Taint-Style Vulnerabilities in Web Applications (Technical Report). To appear, 2007.

[41] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *International Conference on World Wide Web (WWW)*, 2006.

[42] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1973.

[43] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *ACM Symposium on Applied Computing (SAC)*, 2006.

[44] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.

[45] William Landi and Barbara G. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[46] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Usenix Security Symposium*, 2001.

[47] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization Across Object Abstraction. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[48] V. Benjamin Livshits and Monica S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Usenix Security Symposium*, 2005.

[49] Florian Martin. *Generating Program Analyzers.* PhD thesis, University of Saarland, 1999.

[50] Yasuhiko Minamide. Static Approximation of Dynamically Generated Web Pages. In *International Conference on World Wide Web (WWW)*, 2005.

[51] ModSecurity. `http://www.modsecurity.org/`.

[52] Mehryar Mohri and Richard Sproat. An Efficient Compiler for Weighted Rewrite Rules. In *Annual Meeting on Association for Computational Linguistics*, 1996.

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

[54] Neohapsis Archives (BugTraq): Vulnerabilities in SquirrelMail. `http://archives.neohapsis.com/archives/bugtraq/2002-01/0310.html`.

[55] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *IFIP International Information Security Conference*, 2005.

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

[57] PAG/WWW: Static Program Analysis. http://www.program-analysis.com.

[58] Persistent Client State: HTTP Cookies. http://wp.netscape.com/newsref/std/cookie_spec.html.

[59] PHP: Hypertext Preprocessor. http://www.php.net.

[60] PHP Manual. http://www.php.net/manual/en.

[61] PhpParser. http://www.infosys.tuwien.ac.at/staff/enji/PhpParser.html.

[62] PHP Session Security. http://www.webkreator.com/php/configuration/php-session-security.html.

[63] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[64] RFC 2616, Security Considerations. http://www.w3.org/Protocols/rfc2616/rfc2616-sec15.html.

[65] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. In *IEEE Journal on Selected Areas in Communications*, pages 5 – 19, January 2003.

[66] Thomas Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. http://www.securenet.de/papers/Session_Riding.pdf.

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

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

[69] Sed: Stream Editor. http://sed.sourceforge.net/.

[70] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Usenix Security Symposium*, 2001.

[71] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis.* in Program Flow Analysis: Theory and Applications. 1981.

[72] Chris Shiflett. Foiling Cross-Site Attacks. http://www.securityfocus.com/archive/1/191390.

[73] Chris Shiflett. PHP Security. In *O'Reilly Open Source Convention*, 2004.

[74] SourceForge. http://sourceforge.net/.

[75] SquirrelMail. http://www.squirrelmail.org/.

[76] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

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

[78] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.

[79] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, , and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2007.

[80] Peter W. Cross-Site Request Forgeries. http://www.securityfocus.com/archive/1/191390.

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

[82] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[83] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[84] Wikipedia. Hasse diagram. http://en.wikipedia.org/wiki/Hasse_diagram.

[85] Robert P. Wilson and Monica S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[86] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Usenix Security Symposium*, 2006.

[87] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Usenix Security Symposium*, 2002.

List of Figures

List of Tables

Appendix A

Lattices

This chapter provides an introduction to a number of concepts in lattice theory, especially to those needed for understanding data flow analysis. It starts with the basics, requires only little previous knowledge, and tries to illustrate the topics with concrete examples. Most of these examples will refer to the following simple set of natural numbers from 1 to 3: {1, 2, 3}.

A.1 Binary Relations

A **binary relation** R over a set is quite simple: When receiving two elements from this set as input, it returns "true" or "false" as output. For instance, consider the relation "lesser than or equal to" ($\leq$) over the set {1,2,3}. If this relation receives the input (1,2), it returns "true" (since $1 \leq 2$). For (3,2), it returns "false".

A.2 Partial Order Relations

Partial order relations belong to the class of binary relations, but additionally have the following special characteristics:

- Reflexivity: If we take an element a of the underlying set, then the relation for the pair (a,a) must be true. For instance, the pair (1, 1) returns "true" for the relation $\leq$. As a shorthand, we write "aRa" for "relation R returns 'true' for the input (a,a)".

- Antisymmetry: If aRb and bRa, then a must be equal to b. For example, $1 \leq x$ and $x \leq 1$ can only be true if $x = 1$.

- Transitivity: From aRb and bRc, it follows that aRc (e.g. $1 \leq 2$ and $2 \leq 3$ leads to $1 \leq 3$).

As we saw in the examples for reflexivity, antisymmetry and transitivity, the relation $\leq$ satisfies all these properties, and therefore, is a partial order relation.

A.3 Partially Ordered Sets

A set together with a partial order relation is called **partially ordered set** (short name: **poset**). In this context, the word "partial" denotes the fact that there *does not need to be an order for all pairs* of elements from the underlying set. If there *is* an order for all pairs, we are dealing with a special form of partially ordered sets, namely the **totally ordered set**. In fact, the set {1,2,3} with the relation $\leq$, or in short written as $(\{1, 2, 3\}, \leq)$, is a totally ordered set. For demonstrating a partially ordered set which is not a totally ordered set, first consider the **powerset** of {1,2,3}. This is a set containing all subsets of {1,2,3}: { {1,2,3}, {1,2}, {1,3}, {2,3}, {1}, {2}, {3}, {} }. Note that a powerset also includes the empty set, since the empty set is a subset of every set. The notation for the powerset of some set S is $\mathcal{P}(S)$ or 2^S. Now we define a partial order on this set, namely the subset inclusion $\subseteq$. Here are some pairs for which this relation is true:

({1}, {1,2})	because {1} $\subseteq$ {1,2}
({2}, {1,2,3})	analogous
({2,3}, {1,2,3})	analogous
({}, {1})	because the empty set is subset of every set

This powerset is not totally ordered, but only partially ordered. The reason is that it contains pairs of elements for which no order exists, such as:

({1}, {2})	because neither {1} $\subseteq$ {2} nor {2} $\subseteq$ {1}
({1,2}, {2,3})	analogous

Up to this point, we have already presented two partial order relations: $\leq$ and $\subseteq$. From now on, we will write $\sqsubseteq$ to denote some arbitrary partial order relation.

Posets can be depicted graphically. We can represent the poset elements as nodes and the relation between pairs of elements as directed edges (i.e., edges with an arrow at one end). However, this method is rather awkward because the number of edges increases rapidly with the number of elements. A more elegant way is to use a "Hasse diagram", which makes some of the explicit information implicit and hence, becomes smaller. More precisely, it does the following:

- Reflexive edges (i.e., edges whose source and target nodes are identical) are omitted. Since *all* elements of a poset have a reflexive relation, this omission does not lead to a loss of information.

- Transitive edges are omitted, i.e., if there is an edge from {1} to {1,2} and one from {1,2} to {1,2,3}, there is no need to draw an additional transitive edge from {1} to {1,2,3}.

- By convention, all edges are directed "upwards", so it is not necessary to add arrows to the edges.

Figure A.1 illustrates the Hasse diagram for the poset $(\{1, 2, 3\}, \subseteq)$.

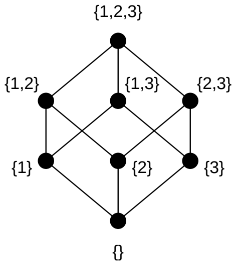

Figure A.1: A Hasse diagram.

A.4 Bounds

An **upper bound** u of two elements a and b in a poset is defined in the following way: Both $a \sqsubseteq u$ and $b \sqsubseteq u$ must be true. For example, the elements {1,2} and {1,3} have the upper bound {1,2,3}, which can be easily seen in the Hasse diagram in Figure A.1. The elements {1} and {1,2} have the following upper bounds: {1,2}, {1,2,3}. This shows that there can be multiple upper bounds, and that upper bounds do not have to be different from the elements they are computed for.

The **least upper bound** (also called **supremum** or **join**) is the upper bound that is $\sqsubseteq$ all other upper bounds. For example, the least upper bound of {1} and {1,2} (written as $\{1\} \sqcup \{1,2\}$) is {1,2}, and $\{1,2\} \sqcup \{1,3\} = \{1,2,3\}$. The least upper bound is always unique (if it exists), but does not have to be different from the elements it is computed for. The notions of **lower bounds** and **greatest lower bounds** (also known as **infimum** or **meet**) are defined analogously.

Upper and lower bounds (as well as their "least" and "greatest" variants) can also be computed for multiple elements (i.e., sets of elements) instead of only for two elements. For example, the least upper bound of {1}, {2}, and {3} is {1,2,3}. In short: $\bigsqcup(\{1\},\{2\},\{3\}) = \{1,2,3\}$

The **greatest element** of a set of elements inside a poset has the following property: It must be among the given set of elements, and all other elements in the given set must be $\sqsubseteq$ this greatest element. For example, the greatest element of the set {{1}, {1,2}, {1,2,3}} is {1,2,3}. The **least element** can be defined analogously to the greatest element.

Note that there also exist **maximal** and **minimal** elements, which are different from greatest and least elements. However, these notions are not necessary for the understanding of lattices, and are skipped in this introduction.

A.5 Lattices

A **lattice** is a poset in which *all nonempty, finite* subsets have both a least upper bound and a greatest lower bound. Infinite subsets (as opposed to finite subsets) will be discussed later in this section. The powerset example presented above forms a lattice: Regardless of which and how many elements from the underlying set are considered, it is always possible to compute a least upper bound as well

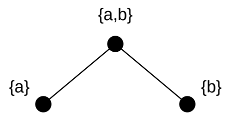

Figure A.2: A poset that is not a lattice.

as a greatest lower bound for these elements. For a better understanding of the concept of lattices, it is useful to see an example for a poset that is *not* a lattice, which is shown in Figure A.2. This figure depicts a poset with three elements ordered by subset inclusion. Since there is no greatest lower bound for the elements a and b, this poset does not satisfy the lattice condition.

A **complete lattice** is a poset in which *all* subsets have both a least upper bound and a greatest lower bound. Note the difference between complete lattices and "normal" lattices: While for complete lattices, the bounds condition has to be satisfied by all subsets, normal lattices have to satisfy it only for nonempty finite subsets. It can be shown that all lattices with a finite number of elements are always complete lattices. Hence, the difference can only be demonstrated with a lattice that has an infinite number of elements. For instance, consider a poset with all integers ($\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$) as its elements and the relation $\leq$ as partial order relation. Obviously, this poset is infinite, and it satisfies the normal lattice condition. However, one of the subsets in this lattice is the set $\mathbb{Z}$ itself. Since neither a least upper bound nor a greatest lower bound can be computed for this subset, the condition for a complete lattice is not satisfied. This normal lattice can be turned into a complete lattice by adding the elements ∞ and $-\infty$ to the underlying set. Intuitively, this means that a complete lattice must always have a greatest element (the **top element**, written as $\top$) as well as a least element (the **bottom element**, written as $\bot$).

Note that the above definition of a complete lattice can be cut down to requiring only a least upper bound for all subsets, or only a greatest lower bound for all subsets. The reason is that one condition automatically implies the other.

One of the least intuitive aspects of complete lattices has to do with the **empty subset** (written as $\emptyset$). First, it is important not to confuse the empty subset of a lattice with a simple element of a lattice. For instance, consider a powerset lattice with the elements {{}, {1}, {2}, {1,2}}. One of the elements is the empty set {}, but this is *not* the empty subset $\emptyset$ of this lattice. The empty subset corresponds to considering *nothing* of the lattice, not even the element {}. A second difficulty with regard to complete lattices is understanding the following equations. These equations are correct, even though this might not be apparent at first glance:

$$\bigsqcup \emptyset = \bot$$

$$\bigsqcap \emptyset = \top$$

Until now, we have already computed the least upper bounds for lattice subsets consisting of two or more lattice elements. The least upper bound of a single lattice element is straightforward, as it is identical to the element itself. But in the first of the above equations, the least upper bound of *no element* is computed. The key to understanding the result of this computation is the interesting fact that virtually *every* statement about *all elements of* $\emptyset$ is true, simply because these elements do not exist. For instance, it is possible to say "every element of $\emptyset$ is smaller than 42", and it would be true because there exists no element that could refute this statement. Therefore, it is also valid to say that every element of $\emptyset$ is $\sqsubseteq$ any element of the complete lattice L. This means that all elements of L are upper bounds of $\emptyset$. Hence, the *least* upper bound of $\emptyset$ is the *least* element of L, denoted as $\bot$. Using an analogous argumentation, it follows that the greatest lower bound of $\emptyset$ is $\top$. Note that $\emptyset$ is the only subset of L for which the greatest lower bound is *larger* than the least upper bound (this is the most confusing detail in this respect). Besides, all this also leads to the insight that complete lattices must always have one or more elements. Otherwise, the least upper bound of the empty subset would not exist - but according to the definition of a complete lattice, it has to exist. In the case of normal lattices, issues related to $\emptyset$ are irrelevant, since their definition only refers to "nonempty" subsets.

In older literature, the term **semilattice** is used occasionally. As the name implies, it resembles a normal lattice, with the difference that only one of the two demands has to be satisfied: It is a poset in which all nonempty finite subsets have a least upper bound *or* a greatest lower bound. Compared to the normal lattice definition, the "and" was replaced by an "or". Depending on which of the two conditions a semilattice satisfies, it is either called a **join-semilattice**, or a **meet-semilattice**.

List of Publications

Cross Site Scripting Prevention
with Dynamic Data Tainting and Static Analysis.
Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna.
Network and Distributed System Security Symposium (NDSS).
San Diego, CA, USA, February 2007.

Preventing Cross Site Request Forgery Attacks.
Nenad Jovanovic, Engin Kirda, and Christopher Kruegel.
IEEE International Conference on Security and Privacy in Communication Networks (SecureComm).
Baltimore, MD, USA, August 2006.

Precise Alias Analysis for Static Detection of Web Application Vulnerabilities.
Nenad Jovanovic, Christopher Kruegel, and Engin Kirda.
ACM Workshop on Programming Languages and Analysis for Security.
Ottawa, Canada, June 2006.

Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper).
Nenad Jovanovic, Christopher Kruegel, and Engin Kirda.
2006 IEEE Symposium on Security and Privacy.
Oakland, CA, USA, May 2006.

SecuBat: A Web Vulnerability Scanner.
Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic.
The 15th International World Wide Web Conference (WWW 2006).
Edinburgh, Scotland, May 2006.

Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks.
Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic.
The 21st ACM Symposium on Applied Computing (SAC 2006), Security Track.
Dijon, France, April 2006.

Curriculum Vitae (German) - Lebenslauf

14.06.1982	geboren in Baden bei Wien (Österreich)
09/1992 - 07/2000	Bundesrealgymnasium Biondekgasse in Baden
10/2000 - 11/2004	Studium der Wirtschaftsinformatik an der TU Wien
08.11.2004	Verleihung des Grades Mag.rer.soc.oec. durch die TU Wien, Titel der Diplomarbeit: “Entwicklung eines webbasierten Instituts-Informationssystems unter besonderer Berücksichtigung der Websicherheit”
11/2004 - 10/2007	Doktoratsstudium an der TU Wien
08/2005 - 08/2007	Projektassistent am Institut für Informationssysteme, TU Wien