Web Application Security—Past, Present, and Future *

Yao-Wen Huang† and D. T. Lee‡

Institute of Information Science, Academia Sinica
Nankang 115, Taipei, Taiwan

{ywhuang, dtlee}@iis.sinica.edu.tw

Abstract

Web application security remains a major roadblock to universal acceptance of the Web for many kinds of online transactions, especially since the recent sharp increase in remotely exploitable vulnerabilities has been attributed to Web application bugs. In software engineering, software testing is an established and well-researched process for improving software quality. Recently formal verification tools have also shown success in discovering vulnerabilities in C programs. In this chapter we shall discuss how to apply software testing and verification algorithms to Web applications and improve their security attributes. Two of the most common Web application vulnerabilities that are known to date are script injection, e.g., SQL injection, and cross-site scripting (XSS). We will formalize these vulnerabilities as problems related to information flow security—a conventional topic in security research. Using this formalization, we then present two tools, WAVES (Web Application Vulnerability and Error Scanner) and WebSSARI (Web Application Security via Static Analysis and Runtime Inspection), which respectively utilize software testing and verification to deal in particular with script injection and XSS and address in general the Web application security problems. Finally we will present some results obtained by applying these tools to real-world Web applications that are in use today, and give some suggestions about the future research direction in this area.

1. INTRODUCTION

As World Wide Web usage expands to cover a greater number of B2B (business-to-business), B2C (business-to-client), healthcare, and e-government services, the reliability and security of Web applications has become an increasingly important concern. In a Symantec analysis report of network-based attacks, known vulnerabilities, and malicious code recorded throughout 2003 [49], eight of the top ten attacks were associated with Web applications; and the report also stated that port 80 was the most frequently attacked TCP port. In addition to holding Web applications responsible for the sharp increase in moderately severe vulnerabilities found in 2003, the authors of the report also suggested that Web application vulnerabilities were by far the easiest to exploit.

Web application insecurity is attributed to several factors. Firstly, the Web, which was initially designed as a data-delivery platform, has quickly evolved into a complex application platform on top of which more and more sophisticated applications have been developed. As a result, Web specifications have grown rapidly to meet rising demands, and browsers and Web-development languages fought a “feature war” to win market share. Unfortunately, security issues have been left as an afterthought. The fast-expanded features did help Web growth; however, many security side effects they induced have become today’s major concern for Web adoption. Secondly, since software vendors are becoming more adept at writing secure code and developing and distributing patches to counter traditional forms of attack (e.g., buffer overflows), hackers are increasingly targeting Web applications. Web application vulnerabilities are hard to eliminate because most Web applications a) go through rapid development phases with extremely short turnaround time, and b) are developed in-house by corporate MIS engineers, most of whom have less training and experience in secure software development compared to engineers at IBM, Sun, Microsoft, and other large software firms. Lastly, current technologies such as anti-virus software and network firewalls offer comparatively secure protection at the host and network levels, but not at the application level [24]. When network and host-level entry points are relatively secure, the public interfaces to Web applications become the focus or targets of attacks [68] [24].

Two of the most common Web application vulnerabilities are script injection (e.g., SQL injection) and cross-site

† *also with the Dept of Electrical Engineering, National Taiwan University*

‡ *also with the Dept of Computer Science and Information Engineering, National Taiwan University*

* *This work was supported in part by the National Science Council under the Grants NSC-93-2213-E-001-013, NSC-93-2422-H-001-0001, and NSC-93-2752-E-002-005-PAE*

scripting (XSS). In this chapter we shall first provide a brief description of XSS and script injection vulnerabilities. The reader is referred to Scott and Sharp [98] [99], Curphey et al. [24], and Meier et al. [68] for more details. We then describe possible automated approaches to eliminating or at least detecting such vulnerabilities. Finally we will give some concluding remarks and present a few possible avenues for future work in this area.

1.1 Cross-Site Scripting (XSS)

On Feb 2, 2000, CERT Coordination Center issued an advisory [18] on "cross-site scripting" (XSS) attacks on Web applications. This hard-to-eliminate threat soon drew the attention and spawned active discussions among security researchers [79]. Despite the efforts of researchers in the private sector and academia to promote developer awareness and to develop tools to eliminate XSS attacks, hackers are still using them to exploit Web applications. A study by Ohmaki (2002) [80] found that almost 80 percent of all e-commerce sites in Japan were still vulnerable to XSS. A search on Google News (http://news.google.com) for XSS advisories on newly discovered XSS vulnerabilities within the month of March 2004 alone yielded 24 reports. Among these were confirmed vulnerabilities in Microsoft Hotmail [108] and Yahoo! Mail [62], both of which are popular web-based email services. Figure 1 gives an example of an XSS.

```
$nick=$_GET['nick'];
echo "Welcome, ".$nick."!"
```

Figure 1. Example of an XSS vulnerability.

Values for the variable $nick come from HTTP requests and are used to construct HTML output sent to the user. An example of an attacking URL would be:

http://www.target.com/default.php?nick=<script>malicious_script();</script>

Attackers must find ways to make victims open this URL. One strategy is to send an e-mail containing a piece of Javascript that secretly launches a hidden browser window to open this URL. Another is to embed the same Javascript inside a Web page, and when victims open the page, the script executes and secretly opens the URL. Once the PHP code shown in Figure 1 receives an HTTP request for the URL, it generates the compromised HTML output shown in Figure 2.

```
Welcome, <script>malicious_script();</script>!
```

Figure 2. Compromised HTML output.

In this strategy, the compromised output contains malicious script prepared by an attacker and delivered on behalf of a Web server. HTML output integrity is hence broken and the Javascript Same Origin Policy [69] [78] is violated. Since the malicious script is delivered on behalf of the Web server, it is granted the same trust level as the Web server, which at minimum allows the script to read user cookies set by that server. This often reveals passwords or allows for session hijacking. Furthermore, if the Web server is registered in the Trusted Domain of the victim's browser, other rights (e.g., local file system access) may be granted as well.

1.2 SQL Injection

Considered more severe than XSS, SQL injection vulnerabilities occur when untrusted values are used to construct SQL commands, resulting in the execution of arbitrary SQL commands given by an attacker. Figure 3 shows an example.

```
$sql="INSERT INTO client_log  VALUES('$HTTP_REFERER');";
mysql_query($sql);
```

Figure 3. Example of a SQL injection vulnerability.

In Figure 3, $HTTP_REFERER is used to construct a SQL command. The referrer field of an HTTP request is an untrusted value given by the HTTP client; an attacker can set the field to:

```
'); TRUNCATE TABLE client_log
```

This will cause the code in Figure 3 to construct the $sql variable as:

```
INSERT INTO client_log VALUES(''); TRUNCATE TABLE client_log;
```

Table "client_log" will be emptied when this SQL command is executed. This technique, which allows for the arbitrary manipulation of backend database, is responsible for the majority of successful Web application attacks.

1.3 General Script Injection

General script injection vulnerabilities are considered the most severe of the three types discussed in this chapter. They occur when untrusted data is used to call functions that manipulate system resources (e.g., in PHP: fopen(), rename(), copy(), unlink(), etc) or processes (e.g., exec()). Figure 4 presents a simplified version of a general script injection vulnerability. The HTTP request variable "df" is used as an argument to call fopen(), which allows arbitrary files to be opened. A subsequent code section may deliver the opened file to the HTTP client, which allows attackers to download arbitrary files.

```
$download_file  = $_POST['df'];
if($_POST['action'] == 'download')  $fp=fopen($download_file,'rb');
```

Figure 4. Example of a general script injection vulnerability.

A more severe example of this vulnerability type is shown in Figure 5.

```
exec("validate_user.exe $_POST['user'] $_POST['pass']");
```

Figure 5. A general script injection bug found in *PHP Surveyor*.

The intent for this code is to execute the validate_user.exe program in order to validate user accounts and passwords. However, since the "user" and "pass" variables are untrustworthy, the code permits the execution of arbitrary system commands. For instance, a malicious user can send an HTTP request with `user="x y; NET USER foo /ADD"` and `pass=""`

As a result, the actual command becomes:

```
Validate_user.exe x y; NET USER foo /ADD
```

This results in creation of new user "foo" with logon rights.

2. CURRENT COUNTERMEASURES

In this section we will discuss current countermeasures or approaches to ensuring Web application security. Scott and Sharp [98] [99] have asserted that Web application vulnerabilities are a) inherent in Web application programs; and b) independent of the technology in which the application in question is implemented, the security of the Web server, and the back-end database. An intuitive solution to Web application security is to increase the awareness of secure coding practices during the code development and implementation phase. Recently, the Open Web Application Security Project (OWASP), an open source community dedicated to promoting Web application security, released a list of the "Top Ten Most Critical Web Application Security Vulnerabilities" [82]. Many organizations (including the United States Federal Trade Commission [38]) have recommended the report as a "best practice" for Web application development. VISA referenced the OWASP report in their Cardholder Information Security Program (CISP), and now requires that all custom code be reviewed by knowledgeable reviewers before being put into production [109]. These actions suggest the growth of a security auditing process—perhaps inevitable in light of the errors that even experienced programmers tend to make [50]. Arguably, vulnerabilities are less severe and easier to fix if they are discovered during or very soon after the development stage. However, the process of code auditing by reviewers who are competent enough to detect vulnerabilities is time-consuming and costly [23], and there is no guarantee that such reviews are complete in that they will find every possible flaw in systems containing millions of lines of code. With today's Web applications being developed and constructed by components from sources of different trust levels (e.g., in-house, out-sourced, commercial-off-the-shelf, open-source), there is a serious need for automated mechanisms.

Researchers have proposed a broad range of automated measures against XSS attacks. According to a) the development stage at which they are adopted and b) their underlying technology, these measures can be categorized into four categories—protection, testing, verification, and blended. Table 1 shows a comparison of each category's strengths and drawbacks.

	Stage deployed	Immediate protection	Vulnerability identification	Runtime overhead	Side effects	Source required	Examples
Protection	Production	Yes	No	Yes	No	No	AppShield [95], InterDo [60]
Testing	Production / Development	No	Yes	No	Yes	No	WAVES [51], AppScan [94], WebInspect [104], ScanDo [60]
Verification	Development	No	Yes	No	No	Yes	RATS [102]
Blended	Production /	Yes	Yes	Yes	No	Yes	WebSSARI [53] [54]

	Development						

Table 1—A comparison of the three different strategies for Web application security

2.1 Protection Mechanisms

Installed at the deployment phase and capable of offering immediate security assurance, protection mechanisms are the most widely-adopted solution for Web application security. However, though protective technologies such as anti-virus software, network firewalls and IDSs (intrusion detection systems) offer comparatively secure protection at the host and network levels, application-level [24] protection technologies are still in their infancies. Park and Sandhu's cookie-securing mechanism can be adopted to eliminate XSS, but it requires explicit modifications to existing Web applications. Scott and Sharp [98] [99] proposed the use of a gateway that filters invalid and malicious inputs at the application level; Sanctum's AppShield [95], Kavado's InterDo [60], and a number of commercial products now offer similar strategies. Most of the leading firewall vendors are also using deep packet inspection [31] technologies in their attempts to filter application-level traffic. According to a recent Gartner report [105], those that don't offer application-level protection will eventually "face extinction."

Although application-level firewalls offer immediate assurance of Web application security, they have at least three drawbacks: a) they require careful configuration [16], b) they blindly protect against unpredicted behavior without investigating the actual defects that compromise quality, and c) they induce runtime overhead.

2.2 Formalizing Web Application Vulnerabilities for Testing and Verification

Adopted during the development phase, software testing and verification are two established technologies for improving software quality. Though incapable of offering immediate security assurance, the two technologies can assess software quality and identify defects. To understand how they can be applied to Web applications, we have to first formally model Web application vulnerabilities. The primary objectives of information security systems are to protect confidentiality, integrity, and availability [96]. From the examples described in Section 1, it is obvious that for Web applications, compromises in integrity are the main causes of compromises in confidentiality and availability. The relationship is illustrated in Figure 6. When untrusted data is used to construct trusted output without sanitization, violations in data integrity occur, leading to escalations in access rights that result in availability and confidentiality compromises.

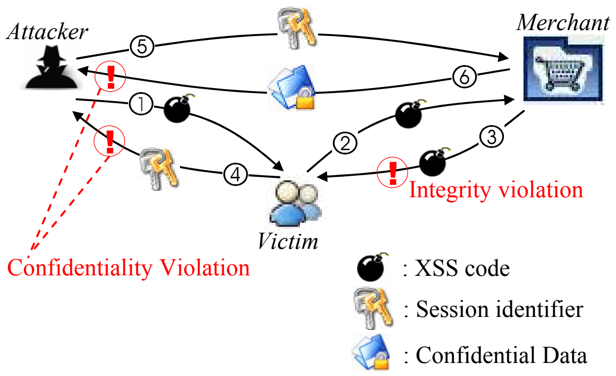


Figure 6. Web application vulnerabilities result from insecure information flow, as illustrated using XSS.

Both software testing and verification techniques can be used to identify illegal information flow—specifically, to identify violations of Web application *noninterference* [43] policies. We first make the following assumptions:

Assumption 1: All data sent by Web clients in the form of HTTP requests should be considered untrustworthy.

Assumption 2: All data local to a Web application are secure.

Assumption 3: Tainted data can be made secure with appropriate processing.

Based on these assumptions we then define the following security policies:

Policy 1: Tainted data must not be used in HTTP response construction.

Policy 2: Tainted data must not be written into local Web application storage.

Policy 3: Tainted data must not be used in system command construction.

Assumption 1 says that all data sent by Web clients (in the form of HTTP requests) should be considered untrustworthy. A majority of Web application security flaws result when this assumption is ignored or neglected. The Web uses a sessionless protocol in which each URL retrieval is considered an independent TCP session, which is established when the HTTP request is sent and terminated after a response is retrieved. Many transaction types (e.g., those that support user logins) clearly require session support. In order to keep track of sessions, Web applications require a client to include a session identifier within an HTTP request. An HTTP request consists of three major parts—the requested URL, form variables (parameters), and cookies. In practice, all three are used in different ways to store session information. Cookies are the most frequently used, followed by hidden form variables and URL requests

To manage sessions, Web applications are written so that browsers include all session information following initial requests that mark the start of a session; and processing HTTP requests entails the retrieval of that information. Even though such information is transferred to the client by the Web application, it should not be considered trustworthy information when it is read back from an HTTP request. The reason is that such information is usually stored without any form of integrity protection (e.g., digital signatures), and is therefore subject to tampering. Using such information to construct HTML output without prior sanitization is considered a Policy 1 violation—the most frequent cause of XSS.

Assumption 2 states that all data local to a Web application should be considered secure. This includes all files read from the file system and data retrieved from the database. According to this assumption, all locally retrieved data are considered trusted, which results in Policy 2, which states that system integrity is considered broken whenever untrustworthy data is written to local storage. Since most applications use client-supplied data to construct output, our model would be too strict without Assumption 3, which states that untrustworthy data can be made trustworthy (e.g., malicious content can be sanitized and problematic characters can be escaped).

XSS vulnerabilities result in Policy 1 or Policy 2 violations. Script injection vulnerabilities such as SQL injection are generally associated with Policy 3 violations.

2.3 Software Testing for Web Application Security

For Web application security, one advantage of software testing over verification is that it considers the runtime behavior of Web applications. It is generally agreed that the massive number of runtime interactions that connect various components is what makes Web application security such a challenging task [59] [98]. Security testing tools for Web applications are commonly referred to as *Web Security Scanners* (WSS). Commercial WSSs include Sanctum's *AppScan* [94], SPI Dynamics' *WebInspect* [104], and Kavado's *ScanDo* [60]. Reviews of these tools can be found in [5], but to our best knowledge no literature exists on their design. Our contribution in this regard is a security assessment framework, for which we have named the *Web Application Vulnerability and Error Scanner*, or *WAVES*. We describe below WSS design challenges and solutions based on our experiences with WAVES.

2.3.1 Testing Model

All WSSs mentioned above are testing platforms posed as outsiders (i.e., as public users) to target applications. This kind of security testing is also referred to as *penetration testing*. They operate according to three constraints:

1. Neither documentation nor source code will be available for the target Web application.
2. Interactions with the target Web applications and observations of their behaviors will be done through their public interfaces, since system-level execution monitoring (e.g., software wrapping, process monitoring, and local files access) is not possible.
3. The testing process must be automated and should not require extensive human participation in test case generation.

Compared with a white-box approach (which requires source code), a black-box approach to security assessment holds many benefits in real-world applications. Consider a government entity that wishes to ensure that all Web sites within a specific network are protected against SQL injection attacks. A black-box security analysis tool can perform an assessment very quickly and produce a useful report identifying vulnerable sites. In white-box testing, analysis of source code provides critical information needed for effective test case generation [87], whereas in black-box testing, an information-gathering approach is to reverse-engineer executable code. WSSs to date take similar approaches to identifying server-side scripts (scripts that read user input and generate output) within Web applications. These scripts constitute a Web application's *data entry points (DEPs)*. Web application interfaces that reveal DEP information include HTML forms and URLs within HTML that point to server-side scripts. In order to enumerate all DEPs of a target Web application, WSSs typically incorporate a webcrawler (also called a softbot or spider) to browse or crawl the target—an approach described in many studies involving Web site analysis (VeriWeb [12], Ricca and Tonella [88] [92]) and a reverse engineering technique (Ricca et al. [89] [90] [91]). From our experiments with WAVES, we learned that ordinary crawling mechanisms normally used for indexing purposes [17] [20] [65] [71] [101] [107] are unsatisfactory in terms of thoroughness. For instance, many pages within Web applications currently contain such dynamic content as Javascripts and DHTML, which cannot be handled by a webcrawler. Other applications emphasize session management, and require the use of cookies to assist navigation mechanisms. Still others require user input prior to navigation. Our tests [51] show that all traditional webcrawlers (which use static parsing and lack script interpretation abilities) tend to skip pages in Web sites that have these features. In both security assessment and fault injection, completeness is an important issue–that is, all data entry points must be correctly identified. Towards this goal, we proposed a "complete crawling" mechanism [51]—a reflection of studies on searching the hidden Web [13] [56] [64] [85] [86].

If each DEP is defined as a program function, then each revelation is the equivalent of a function call site. We define each revelation R of a DEP as a tuple: $R = \{URL, T, Sa\}$, where URL stands for the DEP's URL, T the type of the DEP, and $Sa = \{A_1, A_2, \ldots, A_n\}$ a set of arguments (or parameters) accepted by the DEP. The type of a DEP specifies its functionality. The possible types include searching (tS), authentication (tA), account registration (tR), message posting (tM), and unknown (tU). By combining information on a DEP's URL with the names of its associated HTML forms, the names of its parameters, the names of form entities associated with those parameters, and the adjacent HTML text, WAVES [51] can make a determination of DEP type. Note that form variables are not the only sources of a DEP's input—cookies are also sources of readable input values. Therefore, the set of R's arguments $Sa = S_R \cup S_C$, where $S_R = \{P_1, P_2, \ldots, P_n\}$ is the set of parameters revealed by R, and $S_C = \{C_1, C_2, \ldots, C_n\}$ is the set of cookies contained within the page containing R.

Just as there can be multiple call sites to a program function, there may be multiple revelations of a DEP. In Google, both simple and advanced search forms are submitted to the same server-side script, with the latter submitting more parameters. We defined a DEP D as $\{dURL, dT, dSa\}$. For a set $S_D = \{R_1, R_2, \ldots, R_n\}$ of all collected revelations of the same DEP D, $dURL = R_1.URL = R_2.URL = \ldots = R_n.URL$. D's type dT = Judge_T($R_1.T, R_2.T, \ldots, R_n.T$), where Judge_T is a judgment function that determines a DEP's type, taking into account the types of all its revelations. D's arguments $dSa = R_1.Sa \cup R_2.Sa \cup \ldots \cup R_n.Sa$.

2.3.2 Test Case Generation

Given such a definition, a DEP can be viewed as a program function, with *dURL* being the function name, *dT* the function specification, and *dSa* its arguments. The function output is the generated HTTP response (i.e., HTTP header, cookies, and HTML text). In this respect, testing a DEP is the same as testing a function—test cases are generated according to the function's definitions, functions are called using the test cases, and outputs are collected and analyzed.

Testing for Policy 1 violations involved using our DEP definition to generate test cases containing attack patterns, submitting them to the DEP, and studying the output for signs of the attack pattern. The appearance of an attack pattern in DEP output means that the DEP is using tainted (non-sanitized) data to construct output. The two questions guiding our test case generation were a) What is an appropriate test case size that allows for a thorough testing within an acceptable amount of time? and b) What types of test cases will/will not cause side effects?

In response to the first question, given a DEP D of $dSa = \{A_1, A_2, \ldots, A_n\}$, a naïve approach would be to generate n test cases, each with a malicious value placed in a different argument. For each test case, arguments other than the one containing malicious data would be given arbitrary values. This appears to be a reasonable approach on the surface, but it is subject to a high rate of false negatives because DEPs often execute validation procedures prior to performing their primary tasks. For example, D may use A_1 to construct output without prior sanitization, but at the beginning of its execution it will check A_2 to see if it contains a "@" character, when A_2 represents an email address. In such situations, none of our n test cases would find an error, since they would not cause D to reach its output construction phase. Instead, they would cause D to terminate early and create an error message describing A_2 as an invalid email. However, D would indeed be vulnerable. A human attacker wanting to exploit D could then supply a valid email address and learn that D uses A_1 to construct output without sanitizing it first.

To eliminate this kind of false negatives, we employed a *deep injection* mechanism in WAVES [51]. Using a *negative response extraction (NRE)* technique, the mechanism determines whether or not D uses a validation procedure. The naïve approach is used in the absence of validation. Otherwise, WAVES attempts to use its injection knowledge base to assign valid values to all arguments. Using a trial-and-error strategy, test cases are repeatedly generated and tested in an attempt to identify valid values for all arguments. If successful, then for each of the n test cases, valid values are used for arguments that do not contain malicious data. Otherwise, WAVES degrades to using the naïve approach and generates a message indicating that its test may be subject to a high false negative rate.

2.3.3 Side Effects Elimination

In [51], we acknowledged two serious deficiencies in our original WAVES design—the testing methodology had a potential side effect of causing permanent modifications (or even damage) to the state of the targeted application. For example, for every submission, a DEP D for user registration may add a new user record to a database. If D accepts ten arguments, then to test for a single malicious pattern requires generating ten test cases, with the test pattern placed at a different argument in each test case. But in practice, numerous patterns must be tested in order to provide a decent coverage. And testing for say ten malicious patterns would mean that one hundred meaningless database records would get created.

This potential side effect prevented us from performing large-scale empirical evaluations of WAVES. It should be noted that *AppScan* [94], *InterDo* [60], *WebInspect* [104], and similar commercial and open-source projects have the same drawback. In our subsequent efforts [52], we added three testing modes to WAVES—heavy, relaxed, and safe modes to remedy this drawback. The heavy mode was our original mode; and side effects were simply ignored in the interest of discovering all vulnerabilities. For the two new modes, DEPs were classified according to their types into three disjoint sets S_{safe}, S_{unsafe}, and $S_{unknown}$. $\forall D \in S_{safe}$, $D.T \in$ {tS, tA}; $\forall D \in S_{unsafe}$, $D.T \in$ {tR, tM}; $\forall D \in S_{unknown}$, $D.T$=tU. In both the relaxed and safe modes, DEPs belonging to S_{unsafe} are not tested, and S_{safe} DEPs are tested using the heavy mode. In the relaxed mode, $S_{unknown}$ DEPs are tested using the malicious pattern that is most likely to reveal errors. In safe mode, these are not tested.

2.3.4 Output Observation

After submitting a test case to a DEP, its output (HTTP response) is analyzed to detect any Policy 1 violations. To avoid XSS vulnerabilities, client-submitted data containing <script> HTML tags must be processed prior to being used for output construction. Proper processing entails a) outputting errors that indicate the detection of an attack, and b) removing the tag while still processing the request, and c) encoding the <script> tag so that it is *displayed* rather than *interpreted* by the browser. To help users observe whether such sanitization steps are being taken by a DEP, we have designed test patterns so that the absence of a sanitization routine triggers the execution of a special Javascript by the browser when it renders the DEP output. An example test pattern is shown in Figure 7.

```
<script>alert(“WAVES_TEST_1”);</script>
```

Figure 7. An example of our test pattern for XSS.

As described in Section 2.3.6, Microsoft’s Internet Explorer (IE) was added as a core WAVES component. Accordingly, after submitting the test pattern to a DEP and retrieving its output, it is possible to monitor embedded IE behavior. If IE makes an attempt to display a “WAVES_TEST_1” message box after the response is retrieved, we know that a) the DEP is using one of its arguments to construct output, and b) it did not perform proper sanitization prior to output construction. Such DEPs are considered vulnerable to XSS.

2.3.5 Test Case Reduction

For any DEP accepting *n* arguments, the naïve approach requires $n \times m$ test cases for testing against *m* malicious patterns. To reduce the number of test cases, we modified the test patterns according to the arguments in which the patterns were placed. For example, if placed in the first argument of a DEP, the test pattern shown in Figure 7 will change to:

```
<script>alert(“WAVES_TEST_1_ARG_1”);</script>
```

This allows for the use of IE behavior to identify vulnerable arguments. Using this strategy, we placed modified versions of the same malicious pattern into all arguments of a targeted DEP. This approach requires only $1 \times m = m$ case to be tested against *m* malicious patterns. When two or more malicious patterns appear in the output, the message box events are captured sequentially and vulnerable arguments are identified.

2.3.6 Implementation

WAVES’ system architecture is shown in Figure 8. The webcrawlers act as interfaces between Web applications and software testing mechanisms. Without them we would not be able to apply our testing techniques to Web applications. To make the webcrawlers exhibit the same behaviors as browsers, they were equipped with IE’s Document Object Model (DOM) parser and scripting engine. We chose IE’s engines over others (e.g. Gecko [74] from Mozilla) because IE is the target of most attacks. User interactions with Javascript-created dialog boxes, script error pop-ups, security zone transfer warnings, cookie privacy violation warnings, dialog boxes (e.g. “Save As” and “Open With”), and authentication warnings were all logged but suppressed to ensure continuous webcrawler execution. Note that a subset of the above events is triggered by our test cases or by Web application errors. An error example is a Javascript error event produced by a scripting engine during a runtime interpretation of Javascript code. The webcrawler suppresses the dialog box that is triggered by the event and performs appropriate processing. When an event indicates an error, it logs the event and prepares corresponding entries to generate an assessment report.

When designing the webcrawler, we looked at ways that HTML pages reveal the existence of DEPs or other pages, and came up with the following list:

1. Traditional HTML anchors.
   Ex: <a href = “http://www.google.com“>Google</a>
2. Framesets.
   Ex: <frame src = “http://www.google.com/top_frame.htm”>
3. Meta refresh redirections.
   Ex: <meta http-equiv="refresh"
   Ex: content="0; URL=http://www.google.com">
4. Client-side image maps.
   Ex: <area shape=”rect” href =”http://www.google.com”>
5. Javascript variable anchors.
   Ex: document.write(“\” + LangDir + ”\index.htm”);
6. Javascript new windows and redirections.
   Ex: window.open(“\” + LangDir + ”\index.htm”);
   Ex: window.href = “\” + LangDir + “\index.htm”;
7. Javascript event-generated executions.
   Ex: HierMenus (http://www.webreference.com)
8. Form submissions.

We established a sample site to test several commercial and academic webcrawlers, including Teleport [107], WebSphinx [71], Harvest [17], Larbin [101], Web-Glimpse [65], and Google. None were able to crawl beyond the fourth level of revelation–about one-half of the capability of the WAVES webcrawler. Revelations 5 and 6 were made possible by WAVES’ ability to interpret Javascripts. Revelation 7 also refers to link-revealing Javascripts, but only following an onClick, onMouseOver, or similar user-generated event. WAVES performs an event-generation process to stimulate the behavior of active content. This allows WAVES to detect malicious components and assists in the URL discovery process. During stimulation, Javascripts located within the assigned event handlers of dynamic components are executed, possibly revealing new links. Many current Web sites incorporate DHTML menu systems to aid user navigation. These and similar Web applications contain many links that can only be identified by webcrawlers capable of handling level-7 revelations. Also note that even though the main goal of the injection knowledge manager (IKM) is to produce variable candidates so as to bypass validation procedures, the same knowledge can also be used during the crawling process. When a webcrawler encounters a form, it queries the IKM, and the data produced by the IKM is submitted by the webcrawler to the Web application for deep page discovery.

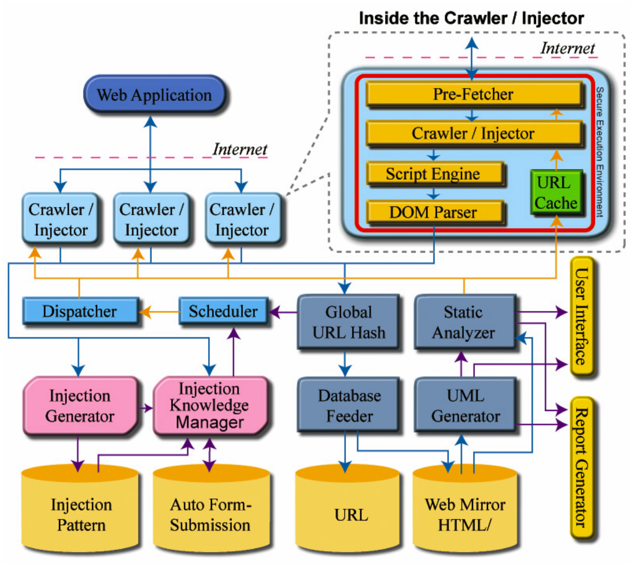


Figure 8. System architecture of WAVES.

In the interest of speed, we implemented a URL hash (in memory) in order to completely eliminate disk access during the crawling process. A separate 100-record cache helped to reduce global bottlenecks at the URL hash. See also Cho and Garcia-Molina [20] for a description of a similar implementation strategy. The database feeder does not insert retrieved information into the underlying database until the crawling is complete. The scheduler is responsible for managing a breadthfirst crawling of targeted URLs; special care has been taken to prevent webcrawlers from inducing harmful impacts on the Web application being tested. The dispatcher directs selected target URLs to the webcrawlers and controls crawler activity. Results from crawling and injections are organized in HTML format by the report generator.

2.3.7 Experimental Result

We evaluated WAVES' DEP discovery ability by comparing its crawling (the number of pages retrieved for a target site) with other webcrawlers. From our tests [51], Teleport [107] proved to be the most thorough of a group of webcrawlers that included WebSphinx [71], Larbin [101], and Web-Glimpse [65]. This may be explained by Teleport's incorporation of both HTML tag parsing and regular expression-matching mechanisms, as well as its ability to statically parse Javascripts and to generate simple form submission patterns for URL discovery. On average, WAVES retrieved 28 percent more pages than Teleport when tested with a total of 14 sites [51]. We attribute the discovery of the extra pages to WAVES' script interpretation and automated form completion capabilities. In case study to evaluate the effectiveness of the different scanning modes we proposed, the heavy mode revealed 80 percent of all errors found by static verification [52]. This shows that our remote, black-box testing approach provides a useful alternative to static analysis when source code and local access to the target Web application is unavailable. The 58.4 percent coverage of the relaxed mode shows that an effective non-detrimental testing is possible. The 55 strictly vulnerable sites identified during a 48-hour relaxed mode scan shows that a) our proposed mechanism for testing insecure information flow can be successfully used to detect XSS, b) non-detrimental testing still yields effective results, and c) XSS still poses a significant threat to today's Web applications. Furthermore, since tools similar to WAVES in many respects are being developed and used by hackers, we note that vulnerable websites can be easily identified by performing controlled "attacks" similar to our experiment with more malicious motivations.

2.4 Software Verification for Web Application Security

Many verification tools are discovering previously unknown vulnerabilities in legacy C programs, raising hopes that the same success can be achieved with Web applications. A major difference between the two efforts is that in C or Java vulnerabilities are introduced by improper control flow, while in Web applications they arise from insecure information flow, against which neither encryption nor traditional Web access control models [83] offer any protection [93]. Sabelfeld and Myers [93] recently published a comprehensive survey on language-based techniques for specifying and enforcing information-flow policies. Among them, sound type systems [111] based on the lattice model of Denning [29] appear most promising. Banerjee and Naumann [9] proposed such a system for a Java-like language, and Pottier and Simonet [84] proposed one for ML. Myers [75] went a step further to provide an actual JIF implementation—a secure information flow verifier for the Java language. However, even though these languages can guarantee secure information flow, many consider them too strict; furthermore, they require considerable effort in terms of additional annotation in order to reduce false positives. Another problem is that most Web applications today are not developed in JIF or Java, but in script languages (e.g., PHP, ASP, Perl, and Python) [55]. Using a type qualifier theory [40], Shankar et al. [103] detected insecure information flow within legacy code with little additional annotation. Using metacompilation-based checkers [46], Ashcraft and Engler [3] were also able to detect insecure information flow in Linux and OpenBSD code without additional annotation. However, checkers are unsound, and both addressed only commonly found insecure information flow problems in C. To our knowledge, no comparable efforts have been made for Web applications, which involve different languages and unique information flow problems.

In contrast to compile-time techniques, run-time protection techniques are attractive because of their accuracy in detecting errors. A typical run-time approach is to instrument code with dynamic guards during the compilation phase. Cowan's Stackguard [22] is representative of this approach; its low overhead and high accuracy has led to its inclusion in a variety of commercial software packages. Immunix Secured Linux 7+ is a commercial distribution of Linux (RedHat 7.0) that has been compiled to incorporate Stackguard instrumentation. Microsoft also includes a feature very similar to Stackguard in its latest release of the Visual C++ .NET compiler [70].

We describe how static and runtime techniques can be used together to establish a holistic and practical approach to ensuring Web application security. We presented here our tool WebSSARI (Web application Security by Static Analysis and Runtime Inspection) [53] [54], which a) statically verifies existing Web application code without any additional annotation effort; and b) after verification, automatically secures potentially vulnerable sections of the code. In order to verify that Policies 1, 2 and 3 hold, WebSSARI incorporates a lattice-based static analysis algorithm derived from type systems and typestate. During the analysis, sections of code considered vulnerable are instrumented with runtime guards, thus securing Web applications in the absence of user intervention. With sufficient annotations, runtime overhead can be reduced to zero. In this section we briefly describe WebSSARI's design and our experiences learned.

2.4.1 Secure Information Flow Research

Type systems have proven useful for specifying and checking program safety properties. By means of programmer-supplied annotations, both proof-carrying codes (PCC) [76] and typed assembly languages (TAL) [73] are designed to provide safety proofs for low-level compiler-generated programs. We also used a type system to verify program security, but we targeted a high-level language, i.e., PHP, and tried to avoid additional annotations.

Many previous software security verification efforts have focused on temporal safety properties related to control flow. Schneider [97] proposed formalizing security properties using *security automata*, which define the legal sequences of program actions. Walker [114] proposed a TAL extension, which uses security policies expressed in Schneider's automata to derive its type system. Jensen, Le Metayer and Thorn [57] proposed using a temporal logic for specifying a program's security properties based on its control flow, and offered a model checking technique for verification. In a similar effort, Chen and Wagner [19] looked for vulnerabilities in real C programs by model checking for violations of a program's *temporal safety properties*. Though their main focus was not on security, Ball and Rajamani [6] adopted a similar approach for their SLAM project and successfully applied it to Windows XP device drivers.

2.4.1.1 Type-Based Analysis

Since vulnerabilities in Web applications are primarily associated with insecure information flow, we focused our effort on ensuring proper information flow rather than control flow. The first widely accepted model for secure information flow was given by Bell and La Padula [11]. They stated two axioms: a) a subject cannot access information classified above its clearance, and b) a subject cannot write to objects classified below its clearance. Their original model only dealt with confidentiality; and Biba [14] is credited with adding the concept of integrity to this model.

Denning [29] established a lattice model for analyzing secure information flow in imperative programming languages based on a program abstraction (similar to Cousot and Cousot's [21] *abstract interpretation*) derived from an *instrumented semantics* of a language. Andrews and Reitman [2] used an axiomatic logic to reformulate Denning's model and developed a compile-time certification method using Hoare's logic. In both cases, soundness was only addressed intuitively (a more formal treatment of Denning's soundness can be found in Mizuno and Schmidt [72]). Orbaek [81] proposed a similar treatment, but addressed the secure information flow problem in terms of data integrity instead of confidentiality. Volpano, Smith and Irvine [111] argued that both works proved soundness with respect to some instrumented semantics whose validity was open to question in that no means was offered for proving that the instrumented semantics correctly reflect information flow within a standard language semantics. To base directly on standard language semantics, Volpano, Smith and Irvine showed that Denning's axioms can be enforced using a type system in which program variables are associated with security classes that allow inter-variable information flow to be statically checked for correctness. Soundness was proven by showing that well-typed programs ensure confidentiality in terms of *noninterference*, a property introduced by Goguen and Meseguer [43] for expressing information flow policies. Recently, fully functional type systems designed to ensure secure information flow have been offered for high-level, strong-typed languages such as ML [84] and Java [75] [9]. Based on Foster et al.'s theory of type qualifiers [40], Shankar et al. [103] used a constraint-based type inference engine for verifying secure information flow in C programs, and detected several format string vulnerabilities in some real C programs of which they were previously unaware.

Type-based approaches to static program analysis are attractive because they prove program correctness without unreasonable computation efforts. Their main drawback is their high false positive rates, which often makes them become impractical for real-world use. Regardless of whether security classes are assigned through manual annotations or through inference rules, in conventional type systems they are statically bound to program variables. It is important to keep in mind that the security class of a variable is a property of its state, and therefore varies at different points or call sites in a program. For example, in Myers' JIF language [75], each program variable is associated with a fixed security label (class). A value assumes the label of the variable in which it is stored. When a value is assigned to a variable, the value loses its original label and assumes the label of the new variable to which it is assigned. Therefore, an assignment causes a re-labeling of the security label of the assigned value. JIF ensures security by only allowing more restrictive re-labeling. However, to precisely capture information flow, values should be associated with fixed security labels, and variables should assume the labels of values they currently store—in other words, assignments should result in the re-labeling of variables rather than values. In JIF and similar type-based systems, variable labels become increasingly restrictive during computation, resulting in high false positive rates. JIF addresses this problem by giving programmers the power to *declassify* variables—that is, to explicitly relax the restrictiveness of variable labels.

2.4.1.2 Dataflow Analysis

False positives resulting from static verification of secure information flow fall into two categories. Class 1 false positives arise from the imprecise approximation of temporal variable properties. The problem described in the preceding paragraph and Doh and Shin's [36] *forward recovery* and *backward recovery* definitions serve as examples. In fact, most of the Denning-based systems suffer from Class 1 errors because the security class of their variables remains constant throughout program execution. Class 2 false positives result from runtime information manipulation or validation. For example, untrusted data can be sanitized before being used, with the original security class no longer applicable. This kind of false positive is more commonly associated with verifications that focus on integrity.

Class 1 errors can be reduced by making approximations of the run-time information flow more precise. Andrews and Reitman [2] first established an approach in which dataflow is semantically characterized in terms of program logic. By applying flow axioms, one can derive flow proofs that specify a program's effect on the information state. This allows the security classes of variables to change during execution, and they argued that their approach captures information flow more precisely than Denning's. Banatre, Bryce, and Le Metayer [8] have offered a comparable approach plus a proof checking method that resembles dataflow analysis techniques associated with optimizing compilers. Joshi and Leino [58] examined various logical forms for representing information flow semantics, leading to a characterization containing Hoare triples. Darvas, Hahnle, and Sands [25] went a step further in offering characterizations in dynamic logic, which allows the use of general-purpose verifications tools (i.e., theorem provers) to analyze secure information flow within deterministic programs.

A similar approach involves flow-sensitive analysis techniques used by optimizing compilers, which have been extensively researched starting from the early works of Allen and Cocke [1] and followed by the works of Hecht and Ullman [47], Graham and Wegman [44], Barth [10], and others. These methods yield more accurate runtime state predictions than the other methods mentioned above. However, flow-sensitivity comes at a price—every branch in a program's control flow doubles the verifier's search space and therefore limits its scalability. ESP, the verification tool recently developed by Das, Lerner, and Seigle [26], is representative of this approach; and is based on the assumption that most program branches do not affect the information flow property that is being checked. Their contribution is distinctive because ESP allows for flow-sensitive verification that scales to large programs. They have also proposed a method called *abstract simulation* to restrict identification and simulation to relevant branch conditions. Unlike ESP, Guyer, Berger, and Lin's [45] approach has a specific security focus. They used the flow-sensitive, context-sensitive, inter-procedural data flow analysis framework provided by their Broadway optimizing compiler to check for format string vulnerabilities of real C programs.

2.4.2 Flow-Sensitive Type-Based Analysis

A third approach emphasizes more accurate or expressive types in type systems. In their trust analysis of C programs, Shankar et al. [103] introduced the concept of type polymorphism in their type qualifier framework, and showed how it can help reduce false positives. Others have considered extending types with state annotations. The most well known approach of this kind is Strom and Yemini's *typestate* [106], which is a refinement of types. According to their definition, an object's type determines a set of allowable operations, while its typestate determines a subset allowable under specific contexts. Because it allows the flow-sensitive tracking of variable states, it serves as a technique applicable to reduce the number of Class 1 errors suffered by type-based information flow systems. Inspired by typestate, DeLine and Fahndrich [28] extended C types in their Vault programming language with predicates (named *type guards*) that describe legal conditions on the use of the type. In other words, types determine valid operations, while type guards determine these operations' valid times of use. In a recent project, Foster et al. [41] extended their original, flow-insensitive type qualifier system for C with flow-sensitive type qualifiers. Using their *Cqual* tool, they demonstrated the effectiveness of their system by discovering a number of previously unknown locking bugs in the Linux kernel.

Interestingly, the authors of ESP [26] (introduced in Section 2.4.1.2), which tracks information flow using dataflow analysis, describe it as "merely a typestate checker for large programs." It appears that as type systems are refined with states and incorporate flow-sensitive checking, fewer differences will exist between type systems and dataflow analysis methods for verifying information flow. Our approach for reducing Class 1 errors is based primarily on typestate.

2.4.2.1 Static Checking

The goal of static *checking* is simply to find software bugs rather than to prove that one does not exist [3]. In other words, checkers are unsound. A pioneering work was that of Bishop and Dilger [15], which checked for "time-of-check-to-time-of-use" (TOCTTOU) race conditions. One recent exciting result is that of Ashcraft and Engler [3], who used their *metacompilation* [46] technique to find over 100 vulnerabilities in Linux and OpenBSD, over 50 of which resulted in kernel patches. The technique makes use of a flow-sensitive, context-sensitive, inter-procedural data flow checking framework that requires no additional annotations. In contrast, Flanagan et al.'s ESC/Java [39] (designed to check the correctness of Java programs) requires additional annotations from programmers.

Most efforts to develop checkers have resulted in publicly available tools [23], including BOON by Wagner et al. [112], RATS by Secure Software [102], FlawFinder by Wheeler [116], PScan by DeKok [27], Splint by Larochelle and Evans [63], and ITS4 by Viega et al. [110]. All these unsound checkers search for specific error patterns. Splint is the only one that requires user annotations. With the exception of ESC/Java, they are all designed for use with C programs.

2.4.2.2 A Comparison

Our algorithm can be described as a sound static verification method and as a holistic method that ensures security in the absence of user intervention. Most type-based static verification methods are considered sound, provided as extensions to existing languages (e.g., Pottier and Simonet [84], Banerjee and Naumann [9], and Myers [75]), and designed to support secure program development (as opposed to verifying existing code). Our work was partly inspired by the type qualifier-based verifier described in [103] (Shankar et al.) and [26] (ESP), both of which offer sound, flow-sensitive, inter-procedural data flow analysis without additional annotations. Broadway [45] offers the same capabilities. Other checkers (e.g., MC [3], RATS [102], and ITS4 [110]) also perform dataflow analysis without additional annotations, but their analyses are considered unsound. And as mentioned in Section 2.4.2.1, most of these checkers (with the exception of RATS) are targeted at C programs, while ours is targeted at PHP scripts. As its name suggests, RATS is simply a Rough Auditing Tool for Security that offers limited checks for defective PHP programming patterns; and in contrast WebSSARI offers a sound information flow analysis. Another difference is that WebSSARI ensures security by inserting runtime guards, while the other tools are limited to providing verification.

WebSSARI, MC and ITS4 are the only approaches that support *automated declassification*, defined as the process of identifying changes in a variable's security class resulting from runtime sanitization or validation. Automated declassification helps reduce the number of Class 2 false positives. MC was designed to detect sections of code that validate user-submitted integers. If the code makes both upper bound and lower bound validations on an untrusted value, it is assumed that validation has been performed; and the security class of the validated value is then changed from untrusted to trusted. This approach is based on the unsound assumption that as long as an untrusted value passes a certain kind of validation, it is actually safe. Therefore, false positives are reduced at the cost of introducing false negatives that compromise verification soundness. In the case of ITS4, its attempt to reduce Class 2 false positives (while detecting C format string vulnerability) involves using lexical analysis to identify sanitization routines based on unsound heuristics.

When verifying information flow in Web applications, one deals with strings instead of integers, and PHP provides standard string sanitization functions. By accepting all string values processed by these functions as trusted, we first reduced a considerable number of Class 2 false positives. For cases in which custom sanitization is provided by the programmer, we proposed *type-aware qualifiers*, which resulted in a more expressive security lattice than the simple tainted-untainted lattice used by other efforts (e.g., Ashcraft and Engler [3] and Shankar et al. [103]), and achieved a further reduction in the number of Class 2 errors.

	Focus	App	Snd	Anno	Lang	Dec
WebSSAR	S. I.F.	Type	Yes	Optional	PHP	Auto
CQual	S. I.F.	Type	Yes	Some	C	Manual
JIF	S. I.F.	Type	Yes	Required	Java	Manual
Vault	Gen. I.F.	Type	Yes	Required	C	Manual
ESP	Gen.	D.A.	Yes	No need	C	None
Broadway	S. I. F.	D.A.	Yes	No need	C	None
MC	S. I.F.	D.A.	No	No need	C	Auto
BOON	S.	D.A.	No	No need	C	None
ESC/Java	Gen.	D.A.	No	Required	Java	Manual
Splint	S.	L.A..	No	Required	C	Manual
ITS4	S.	L.A.	No	No need	C	Auto
MOPS	S.	Modl	Yes	No need	C	None

App—Approach	Snd—Soundness
Anno—Annotation effort	Lang—Supported language
Dec—Declassification support	S.—Focus on security
I.F.—Focus on information flow	Gen.—General verification
Type—Type system	D.A.—Dataflow analysis
L.A.—Lexical analysis	Modl—Model checking

Figure 9. A comparison among related works.

To provide a clear representation of how our efforts compare with those of others, we have defined six criteria for classifying static analyzers: focus of scope, approach, soundness, additional annotation effort, supported language, and declassification support. A comparison based on these criteria is presented in Figure 9.

2.4.2.3 Runtime Protection

In many situations, it is difficult for static analysis to offer satisfactory runtime program state approximation. One strategy is to delay parts of the verification process until runtime. A good example of this practice is Perl's "tainted mode" [113], which ensures system integrity by tracking tainted data submitted by the user at runtime. In a similar manner, Myers [75] also leaves some JIF security class checking operations until runtime. In dynamically typed languages such as Lisp and Scheme, a common approach is to perform runtime type checking for objects whose types have yet to be determined at compile-time. These kinds of dynamic checks are extremely expensive, resulting in the creation of such static optimization techniques as dynamic typing [48] and soft typing [117] to reduce the number of runtime checks.

WebSSARI takes a similar approach—that is, by applying static analysis, it pinpoints code requiring runtime checks and inserts the checks. A similar process is found in Necula, McPeak, and Weimer's *CCured* [77]. Though not specifically focused on security, their scheme combines type inference and run-time checks to ensure type safety for existing C programs. A major difference is that our inserted guards perform sanitization tasks rather than runtime type checking—in other words, we insert sanitization routines in vulnerable sections of code that use untrusted information. If they are inserted at the proper locations, their execution time cannot be considered real overhead because the action is a necessary security check; and WebSSARI will have simply inserted lines of code omitted by a careless (or security-unaware) programmer.

2.4.3 Verification Algorithm

In PHP, which is an imperative, deterministic programming language, sets of functions affect system integrity. For example, `exec()` executes system commands, and `echo()` generates output. These functions must be called with trusted arguments. We refer to such functions as *sensitive functions*; and vulnerabilities will result from *tainted* (untrustworthy) data used as arguments in sensitive function calls. We intuitively derived a trust policy (expressed as a *precondition* of the function), which states the required trust level for each of the function's arguments. We considered all values submitted by a user as tainted, and checked their propagation against a set of predefined trust policies.

2.4.3.1 Information Flow Model

To characterize data trust levels, we followed Denning's [29] model and made the following assumptions:

1. Each variable is associated with a security class (trust level).
2. $T = \{\tau_1, \tau_2, ..., \tau_n\}$ is a finite set of security classes.
3. T is a partially ordered set by $\leq$, which is reflexive, transitive, and anti-symmetric. For $\tau_1, \tau_2 \in T$,

   $\tau_1 = \tau_2$ iff $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$,

   and $\tau_1 < \tau_2$ iff $\tau_1 \leq \tau_2$ and $\tau_1 \neq \tau_2$.
4. T forms a complete lattice with a lower bound $\perp$ such that $\forall \tau \in T, \perp \leq \tau$, and an upper bound $\top$ such that $\forall \tau \in T, \tau \leq \top$.

These assumptions imply that a greatest lower bound operator and a least upper bound operator exist on T. For subset $Y \subseteq T$, let $\sqcap Y$ denote $\top$ if Y is empty and the greatest lower bound of the types in Y, otherwise; let $\sqcup Y$ denote $\perp$ if Y is empty and the least upper bound of the types in Y, otherwise.

To develop an information flow system, we need to provide a method to express the trust levels of variables. Following the lead of Foster et al. [40] and Shankar et al. [103], we extended the existing PHP language with extra *type qualifiers*—a widely-used annotation mechanism for expressing type refinements. When used to annotate a variable, the C type qualifier `const` expresses the constraint that the variable can be initialized but not updated [40]. We used type qualifiers as a means for explicitly associating security classes with variables and functions. In our WebSSARI implementation, we specified preconditions for all sensitive PHP function using type qualifiers. These definitions are stored in a prelude file and loaded by WebSSARI upon startup. Another prelude file contains postconditions for functions that perform sanitization to generate trusted output from tainted input. This serves as a mechanism for automated declassifications. A third prelude file includes annotations (using type qualifiers) of all possible tainted input providers (e.g., $_GET, $_POST, $_REQUEST). Type qualifiers are also used as a means for developers to manually declassify variables. Manual declassification support is important because it allows for manual elimination of false positives, which in turn reduces the number of unnecessary runtime guards, resulting in reduce overhead.

Like Foster and Shankar we perform type inferencing (of security classes) in attempt to eliminate user annotation efforts. In conventional type-based secure information flow systems (e.g., JIF [75]), type inferencing is used as a means to infer the *initial* security class of a variable, and a variable is assumed to be associated with its initial security class throughout the entire program execution. As explained in Section 2.4.1.1, fixed variable security classes induce a large number of false positives. To develop a type system in which variable classes can change and flow-sensitive properties can be considered, we maintain our *type judgments* based on Strom and Yemini's [106] typestate. A type judgment Γ is a set of mapping functions (which map variables to security classes) at a particular program point, and every program point has a unique type judgment. For each variable $x \in \mathrm{dom}(\Gamma)$, there exists a unique mapping, $\Gamma \vdash x : \tau$, and we denote the uniquely mapped type τ of x in Γ as $\Gamma(x)$. To approximate runtime typestate at compile-time, a variable's security class is viewed as a *static most restrictive* class of the variable at each point in the program text. That is, if a variable has a particular class τ_p at a particular program point, then its corresponding execution time data object will have a class that is at most as restrictive as τ_p, regardless of which paths were taken to reach that point. Formally, for a set of type judgments G, we denote $\oplus G$ as the most restrictive type judgment Γ where, for all $x \in \{\mathrm{dom}(\Gamma') \mid \Gamma' \in G\}$, $\Gamma \vdash x : \sqcup Y_x$. $Y_X = \{\Gamma'(x) \mid \Gamma' \in G\}$ is the set of all classes of x mapped in G. When verifying a program at a particular program point, $\Gamma = \oplus G_R$, where G_R represents the set of all possible type judgments, each corresponding to a unique execution-time path that could have been taken to reach that point.

To illustrate this concept, we will use the widely-adopted tainted-untainted (T-U) lattice of security classes (e.g., by BOON [112], Ashcraft and Engler [3], and Shankar et al. [103]) shown in Figure 10. The T-U lattice has only two elements—untainted as its lower bound and tainted as its upper bound. Assume that variable t is tainted and that variables u1 and u2 are untainted. Since exec() requires an untainted argument, for Line 2 of Figures 12 and 13 to typecheck requires that we know the static most restrictive class of X. In other words, we need to know the security class τ_{T-2} that is the most restrictive of all possible runtime classes of X at line 2, regardless of the execution path taken to get there. In line 2 of Figure 12, since X can be either tainted or untainted, $\tau_{T-2} = \sqcup\{\text{tainted,untainted}\} = \text{tainted}$; line 2 therefore triggers a violation. On the other hand, line 2 of Figure13 typechecks.

To preserve the static most restrictive class, rules must be defined for resolving the typestate of variable names. For the sake of simplicity, we adopted the original algorithm proposed by Strom and Yemini [106]. First, we perform flow-sensitive tracking of typestate. Then at execution path merge points (e.g., the beginning of a loop or the end of a conditional statement), we define the typestate of each variable as the least upper bound of the typestates of that same variable on all merging paths. In our defined lattice (Figure 11), the least upper bound operator on a set selects the most restrictive class from the set. Note that while Strom and Yemini originally used typestate to represent the *static invariant* variable property, which requires applying the greatest lower bound operator, for our purpose typestate is used to represent the static most restrictive class, so we need to apply the least upper bound instead.

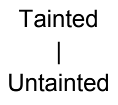


Figure 10. Primitive lattice.

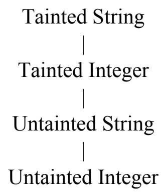


Figure 11. Type-aware lattice.

```
1: if (C) X = t; else X = u1;
2: exec(X);
```

Figure 12. Example A.

```
1: if (C) X = u1; else X = u1;
2: exec(X);
```

Figure 13. Example B

2.4.3.2 Type-Aware Security Classes

The first version of WebSSARI implemented the verification algorithm mentioned above and made use of the T-U lattice. An initial test drive revealed a common type of false positive. Apparently many developers used type casts for sanitization purposes. An example from Obelus Helpdesk is presented in Figure 14. In that example, since `$_POST['index']` is tainted, $i is tainted after line 1. Line 2 therefore does not typecheck, since `echo()` requires untainted values for its argument.

```
1: $i = (int) $_POST['index'];
2: echo "<hidden name = mid value='$i'>"
```

Figure 14. Example of a false positive resulting from a type cast.

Six of the 38 responding developers who also included copies of their intended patches for our review relied on this type of sanitization process. Since all HTTP variables are stored as strings (regardless of their actual type), using a single cast to sanitize certain variables appears to be a common practice. However, the false positive serves as evidence supporting the idea that security classes should be *type-aware*. For example, `echo()` can accept tainted integers without compromising system integrity (i.e., without being vulnerable to XSS). Figure 11 illustrates the type-aware lattice that we incorporated in our second version of WebSSARI. Until now, it has been commonly believed that annotations in type-based security systems should be provided as extensions to be checked separately from the original type system. [40] [41] [103] [39]. In this chapter we are proposing the use of a type-aware lattice model and introducing the idea of *type-aware qualifiers*. Though still checked separately, type refinements (e.g., security classes) are type-aware.

2.4.3.3 Program Abstraction and Type Judgment

When verifying a PHP program, we first use a filter to deconstruct the program into the following abstraction:

$$(commands)\ \ c ::= c_1 ; c_2 \mid x := e \mid e \mid \text{if } e \text{ than } c_1 \text{ else } c_2$$

$$(\exp ression)\ \ e ::= x \mid n \mid e_1 \sim e_2 \mid f(\underline{a})$$

, where x is a variable, *n* is an integer, ~ represents binary operators (e.g., +), $f(\underline{a})$ represents a function call. Commands that do not induce insecure flows are referred to as *valid* commands. The type system maintains a separation between the statically typed and the untyped worlds. To infer types (i.e., security classes) within the untyped world, and to check for command validity in the typed world, we define the following two judgment rules:

(1) Expression typing: $\Gamma \vdash e : \tau$ (2) Command validity: $\Gamma \vdash c$

Commands (which do not produce values) are distinguished from expressions (which do produce values). In these rules, Γ denotes a type judgment, which maps variables to types and also specifies the valid commands. Our type judgment rules are given below:

1.Mapping Rules:

(Initialization) (Operation) (Postcondition)

$$\frac{}{\Gamma \vdash n : \bot} \qquad \frac{x \in dom(\Gamma)}{\Gamma \vdash x : \bot} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \sim e_2 : \tau_1 \sqcup \tau_2} \qquad \frac{\Gamma \vdash f(\underline{a})}{\Gamma \vdash f(\underline{a}) : E_f(X[\underline{a} / \underline{p}])}$$

2.*Checking Rule:*

(Precondition)

$$\frac{f(\underline{a}),\ \Gamma(\underline{a}) \leq \Gamma_f(\underline{p})}{\Gamma \vdash f(\underline{a})}$$

3. Concatenation Rule:

(Concatenation)

$$\frac{\Gamma \vdash c_1 \quad \Gamma' \vdash c_2}{\Gamma, \Gamma' \vdash c_1 ; c_2}$$

4.Updating Rule:

(Assignment) (Restriction)

$$\frac{\Gamma \vdash x : \tau_1 \quad \Gamma \vdash e : \tau_2 \quad x := e}{\Gamma' \vdash x : \tau_1 \sqcup \tau_2 \quad \Gamma' \vdash x := e} \qquad \frac{\Gamma_1 \vdash c_1 \quad \Gamma_2 \vdash c_2 \quad \text{if } e \text{ than } c_1 \text{ else } c_2}{\Gamma' = \Gamma_1 \oplus \Gamma_2 \quad \Gamma' \vdash \text{if } e \text{ than } c_1 \text{ else } c_2}$$

The set of PHP expressions that offer tainted data and the set of sensitive functions are represented as set **I** and set **O**, respectively. To ensure secure information flow, we add the following rule that infers all expressions in I as tainted:

$$\frac{\forall e \in I}{\Gamma \vdash e : ta\operatorname{int}ed} \quad \text{(Tainted Input)}$$

We define preconditions for functions that belong to **O** as the safe sensitive function rule: $\Gamma_f(\underline{p}) = (\text{untainted}, \cdots, \text{untainted})$.

When verifying, we update type judgments according to command sequences and raise an error if any checking rule is violated. If we can derive a type judgment for each program point of the command sequence, we say the command sequence is secure.

2.4.3.4 Soundness

Since we always maintain the static most restrictive type judgment at every program point, a variable's type monotonically increases along the updating sequence. This is an essential property that ensures the soundness of our algorithm. However, PHP is an interpreted "scripting language" that allows for dynamic evaluation. For example, one can write "$$a" to represent a "dynamic variable," whose variable name can be determined only at runtime. To retain soundness, all dynamic variables are considered as tainted. When other kinds of dynamic evaluation exist in the target code, WebSSARI degrades itself to a checker—it still checks for potential vulnerabilities, but outputs a warning message indicating that it cannot guarantee soundness. We do, however, support pointer aliasing by implementing the original solution proposed by Strom and Yemini [106]. We maintain two mappings—an environment and a store. The environment maps the names of variables involved in pointer aliasing to virtual locations, and the store maps locations to security classes. Therefore, when two pointers point to the same storage, we recognize their dereferences as a single value having a single security class. A trust level change in one pointer deference is reflected in the other.

2.4.4 System Implementation

The tool *WebSSARI* was developed to test our approach that extends an existing script language with our proposed type qualifier system. An illustration of WebSSARI's system architecture is presented in Figure 15. A *code walker* consists of a lexer, a parser, an AST (abstract syntax tree) maker, and a *program abstractor*. The program abstractor asks the AST maker to generate a full representation of a PHP program's AST. The AST maker uses the lexer and the parser to perform this task, handling external file inclusions along the way. By traversing the AST, the program abstractor generates a control flow graph (CFG) and a symbol table (ST). The *verification engine* moves through the CFG and references the ST to generate a) type qualifiers for variables (based on the prelude file) and b) preconditions and postconditions for functions. This routine is repeated until no new information is generated. The verification engine then moves through the control flow graph once again, this time performing typestate tracking to determine insecure information flow. It outputs insecure statements (with line numbers and the invalid arguments). For each variable involved in an insecure statement, it inserts a statement that secures the variable by treating it with a sanitization routine. The insertion is made right after the statement that caused the variable to become tainted. Sanitization routines are stored in a prelude file, and users can supply the prelude file with their own routines.

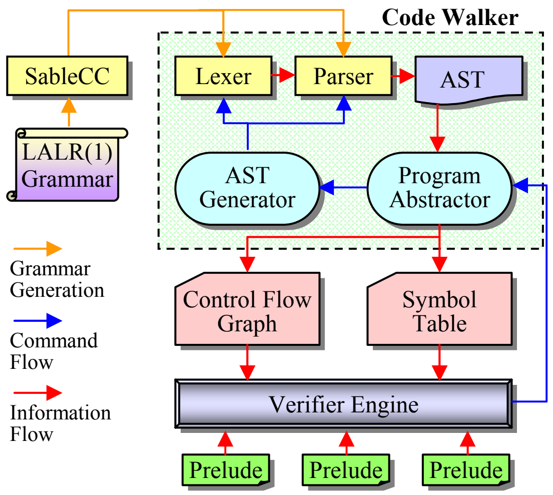
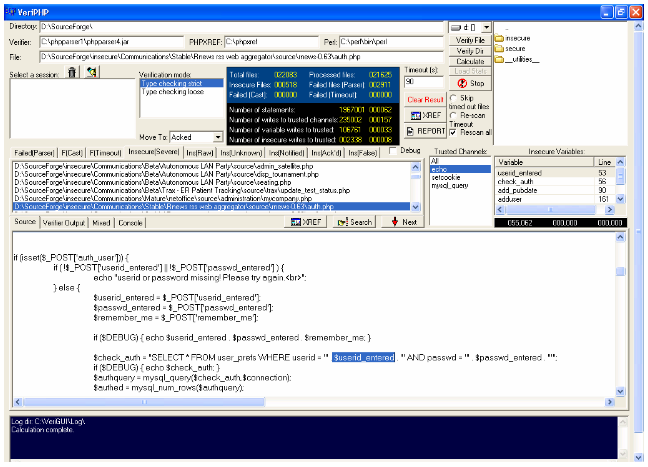


Figure 15. WebSSARI system architecture.

Support for different languages is achieved by providing their corresponding code walker implementations. Since the lexers and parsers can be generated by publicly available compiler generators, providing a code walker for a language breaks down to: a) choosing a compiler generator, and providing it with the language's grammar, b) providing an AST maker, and c) providing a program abstractor. For step a), grammars for widely-used languages (e.g., C, C++, C#, and Java) are already available for widely-used compiler generators such as YACC and SableCC, and for step b), AST makers for different languages should only differ in preprocessing support (e.g., include file handling). However, since we expect considerable differences to exist in the ASTs of various languages, the major focus on providing a code walker implementation for a language is on implementing a program abstractor.

To support verification experiments using tens of thousands of PHP files, we developed a separate GUI featuring batch verification, result analysis, error logging, and report generation. Statistics can be collected based on a single source code file, files of a single project, or files of a group of projects. Vulnerable files are organized according to severity, with general script injection the most severe, SQL injection second, and XSS third. To help users investigate reported vulnerabilities, we added Watts' *PHPXREF* [115] to generate cross-referenced documentation of PHP source files. A screenshot of this GUI is presented in Figure 16.

In this project WebSSARI, we provided a code walker for PHP. We used Gagnon and Hendren's *SableCC* [42], an object-oriented compiler framework for Java. Similar to YACC and other compiler generators, SableCC accepts LALR(1) [30] grammars. No formally written grammar specifications for the PHP language exist, and no studies have been performed on whether PHP's grammar can be fully expressed in LALR(1) form. We used Mandre's [66] LALR(1) PHP grammar for SableCC, which has never been thoroughly tested. The combination of SableCC and Mandre's grammar allowed us to develop a code walker for PHP; however, an initial test drive using approximately 5,000 PHP files revealed deficiencies that caused WebSSARI to reject almost 25 percent of all verified files as grammatically incorrect. With help from Mandre, we were able to reduce that rejection rate to 8 percent in a subsequent test involving 10,000 PHP files.


VeriPHP
Verify File
Verify Dir
Calculate
Load Stats
Stop
Clear Result
XREF
REPORT
Failed(Parser) | F(Cast) | F(Timeout) | Insecure(Severe) | Ins(Raw) | Ins(Unknown) | Ins(Notified) | Ins(Ack'd) | Ins(False)
Source | Verifier Output | Mixed | Console
XREF
Search
Next

Directory: D:\SourceForge\
Verifier: C:\phpparser1\phpparser4.jar PHPXREF: C:\phpxref Perl: C:\perl\bin\perl
File: D:\SourceForge\insecure\Communications\Stable\Rnews rss web aggregator\source\rnews-0.63\auth.php
Select a session:
Verification mode: Type checking strict / Type checking loose
Move To: Acked
Timeout (s): 90
Skip timed out files
Re-scan Timeout
[✓] Rescan all
Debug

```
Total files:       022083    Processed files:       021625
Insecure Files:    000518    Failed files (Parser): 002911
Failed (Cast):     000000    Failed (Timeout):      000000

Number of statements:                     1967001  000062
Number of writes to trusted channels:     235002   000157
Number of variable writes to trusted:     106761   000033
Number of insecure writes to trusted:     002338   000008
```

```
D:\SourceForge\insecure\Communications\Beta\Autonomous LAN Party\source\admin_satellite.php
D:\SourceForge\insecure\Communications\Beta\Autonomous LAN Party\source\disp_tournament.php
D:\SourceForge\insecure\Communications\Beta\Autonomous LAN Party\source\seating.php
D:\SourceForge\insecure\Communications\Beta\Trax - ER Patient Tracking\source\trax\update_test_status.php
D:\SourceForge\insecure\Communications\Mature\netoffice\source\administration\mycompany.php
D:\SourceForge\insecure\Communications\Stable\Rnews rss web aggregator\source\rnews-0.63\auth.php
```

Trusted Channels: All, echo, setcookie, mysql_query

Insecure Variables:

Variable	Line
userid_entered	53
check_auth	56
add_pubdate	90
adduser	161

055,062 000,000 000,000

```
if (isset($_POST['auth_user'])) {
        if ( !$_POST['userid_entered'] || !$_POST['passwd_entered'] ) {
                echo "userid or password missing! Please try again.<br>";
        } else {
                $userid_entered = $_POST['userid_entered'];
                $passwd_entered = $_POST['passwd_entered'];
                $remember_me = $_POST['remember_me'];

                if ($DEBUG) { echo $userid_entered . $passwd_entered . $remember_me; }

                $check_auth = "SELECT * FROM user_prefs WHERE userid = '" . $userid_entered . "' AND passwd = '" . $passwd_entered . "'";
                if ($DEBUG) { echo $check_auth; }
                $authquery = mysql_query($check_auth,$connection);
                $authed = mysql_num_rows($authquery);
```

```
Log dir: C:\VeriGUI\Log\
Calculation complete.
```

Figure 16. A screenshot of the WebSSARI GUI under Windows.

2.4.5 Experimental Results

SourceForge.net [4], the world's largest open source development website, hosts over 70,000 open-source projects for more than 700,000 registered developers. PHP, currently with 7,792 registered projects, clearly outnumbers all other script languages (e.g., Perl, Python, and ASP) for Web application development. SourceForge.net classifies projects according to language, purpose, popularity, and development status (maturity). We identified a sample of 230 projects that reflected a broad variation in terms of language, purpose, popularity, and maturity. We downloaded their sources, tested them with WebSSARI, and manually inspected every report of a security violation. Where true vulnerabilities were identified, we sent email notifications to the developers. Over the five-day test period, we identified 69 projects containing real vulnerabilities; to date, 38 developers have acknowledged our findings and stated that they would provide patches.

We note that in 33 of those 38 projects, the vulnerabilities had simply been overlooked, even though sanitization routines had been adopted in the majority of cases. We also found (from the developers' responses) that some of these projects had vulnerabilities that had already been identified and disclosed prior to the present project. Further inspection of their code revealed that the developers had fixed all previously published vulnerabilities, but failed to identify similar problems that were hidden throughout the code. These observations justify the need for an automated verification tool that can be used repeatedly and routinely.

In all, our WebSSARI scanned 11,848 files consisting of 1,140,091 statements; and 515 files were identified as vulnerable. After four days of manual inspection, we concluded that only 361 files were indeed vulnerable—a false positive rate of 29.9 percent. The number of insecure files dropped to 494 after adding support for type-aware qualifiers, yielding a false positive rate of 26.9 percent. Type-aware qualifiers eliminated the false positive rate by 10.03 percent.

Of the total 1,140,091 statements, 57,404 were associated with making calls to sensitive functions with tainted variables as arguments. WebSSARI identified 863 as insecure. After manual inspection, we concluded that 607 were actually vulnerable. Adding sanitization functions to all 57,404 statements caused 5.03 percent (57,404/1,140,091) of the 1,140,091 statements to be instrumented with dynamic guards, thus inducing overhead. After static analysis, the number of statements requiring dynamic sanitization was reduced to 863—a difference of 98.4 percent. As stated in Section 2.4.2.3, this instrumentation for vulnerable statements cannot be considered overhead because it simply adds code omitted by the programmer. Since only 607 statements were actually vulnerable, WebSSARI only caused 0.02 percent of all statements to be instrumented with unnecessary sanitization routines.

Our experiments were conducted using a machine equipped with one Intel Pentium IV 2.0Ghz processor, 256 megabytes of RAM, and a 7,200 RPM IDE hard disk. On average, WebSSARI processed 73.85 statements per second.

3. CONCLUDING REMARKS AND FUTURE WORK

Security remains a major roadblock to universal acceptance of many kinds of online transactions or services made available through the Web. This concern has been attributed to vulnerabilities of Web applications that are remotely exploitable. Many protection mechanisms are available and can offer immediate security assurance, but they induce overhead and do not address the actual software defects. On the other hand, software testing and verification are both common practices for improving software quality. In order to apply existing techniques to Web applications, Web application vulnerabilities must be formalized. In this chapter, we have formalized Web application vulnerabilities as problems involving insecure information flow, which is a conventional topic in security research. Secure information flow research was mostly motivated by confidentiality considerations; however, we have shown that Web application security require more emphasis on data integrity and trust than on confidentiality and availability. Based on our formalization, we then described how software security testing and verification could be applied to Web applications security.

In software testing, researchers and engineers from the private sector have devoted a considerable amount of resources to developing WSSs, but little is known about their design challenges and their potential side effects. Another drawback is that current WSSs (including our original WAVES [51]) focus on SQL injection detection, but are deficient in XSS detection. We addressed these problems by:

1. giving a formal definition of a WSS and a list of design challenges;
2. listing test types that may induce side effects;
3. describing a test case generation process capable of producing a non-detrimental set of test cases;
4. showing how a Web application can be observed from a remote location during testing;
5. defining three modes of remote security auditing, with a focus on potential side effects;
6. conducting an experiment using three different modes (heavy, relaxed and safe modes) and five real-world Web applications to compare differences in their coverage and induced side effects; and
7. conducting an experiment using the relaxed mode to scan random websites.

At least four assessment frameworks for Web application security (WAVES [51], AppScan [94], WebInspect [104], and ScanDo [60]) provide black-boxed testing capability for identifying Web application vulnerabilities. The advantage of testing over protection mechanisms is their ability to assess software quality. However, they have two disadvantages: a) they cannot provide immediate security assurance, and b) they can never guarantee soundness (they can only attempt to identify certain vulnerabilities, but cannot prove that certain vulnerabilities do not exist). By combining runtime mitigation and static verification techniques, WebSSARI demonstrates an approach that retains the advantages and eliminates the disadvantages of preceding efforts. Note that WebSSARI provides immediate protection at a much lower cost than Scott and Sharp's, since validation is restricted to potentially vulnerable sections of code. If it detects the use of untrusted data following correct treatment (e.g., sanitization), the code is left as-is. According to our experiment, WebSSARI only caused 0.02 percent of all statements to be instrumented with unnecessary sanitization routines. In contrast, Scott and Sharp [98] [99] perform unconditional global validation for every bit of user-submitted data without considering the fact that the Web application may have incorporated the same validation routine, thus resulting in unnecessary overhead. If a Web application utilizes HTTPS for traffic encryption, the associated decrypt-validate-encrypt may limit scalability. Furthermore, WebSSARI provides protection in the absence of user intervention, as compared with the user expertise required for Scott and Sharp's approach. Compared to WAVES, WebSSARI offers a sound verification of Web application code. Since verification is performed on source code, it does not require targeted Web applications to be up and running, nor is there any danger of introducing permanent state changes or loss of data.

Compared to language-based approaches such as Myers [75], Banerjee and Naumann [9], and Pottier and Simonet [84], our approach verifies the most commonly used language for Web application programming, and also incorporates support for extending to other languages. In other words, we provide verification for existing applications while others have proposed language frameworks for developing secure software. Their technique of typing variables to fixed classes results in a high false positive rate; while in contrast, we used typestate to perform flow-sensitive tracking that allows security classes of variables to change, resulting in more precise compile-time approximations of runtime states. Comparing flow-sensitive approaches such as Ashcraft and Engler [3] and Shankar et al. [103], we proposed a type-aware lattice model in contrast to their primitive tainted-untainted model. According to our experimental results, the use of this lattice model helped reduce false positives by 10.03 percent. Compared to unsound checkers [3] [39] [112] [102] [116] [27] [63] [110], WebSSARI attempts to provide a sound verification framework.

We note that compared with a white-box approach (which requires source code) such as WebSSARI, a black-box testing approach to security assessment (e.g., WAVES and other WSSs) still holds many benefits in real-world applications. A black-box security analysis tool can perform an assessment very quickly and produce a useful report identifying vulnerable sites. To assure high security standards, white-box testing can be used as a complement.

3.1 Future Research Directions and Open Problems

3.1.1 Protection Mechanisms—anomaly detection or misuse detection?

The primary job of protection mechanisms such as Scott and Sharp's work [98] [99] or commercial application firewalls is to distinguish malicious traffic from normal traffic. In anomaly detection, normal traffic is defined, and those that do not comply with the definitions are considered malicious. Scott and Sharp adopted this approach for every DEP of a Web application, and required that the network administrator supply definitions describing valid parameters for the DEP. HTTP requests to a DEP that do not comply with its definition are considered malicious. Most commercial application firewalls, on the other hand, deploy deep packet inspection [31], which makes use of misuse detection. In misuse detection, a database of malicious patterns ("signatures") is maintained, and every HTTP request is filtered against this signature database to verify the absence of malicious data. Unfortunately, even for *known* attacks, neither anomaly nor misuse detection can guarantee detection. Scott and Sharp's approach asks that administrators specify valid parameters for every DEP. Though this reduces the chance of DEPs being attacked, it does not eliminate all attack possibilities. For example, the definition for an address field may be "it must be a string with length between 20 to 50 characters." A skilled attacker may still be able to exploit the DEP under this restriction—using a cleverly-crafted 20-to-50-character malicious string. On the other hand, filtering against a signature database cannot guarantee detection either. Signature-based detection has proved very successful in the anti-virus technology, because once released by its developer, a virus' executable code is fixed. However, due to the expressiveness and rich features of SQL, a same SQL injection attack can take almost an unlimited number of patterns. A detailed explanation was recently given in Maor and Shulman [67]. Even if all possible attack patterns can be enumerated, real-time filtering would be impractical even with the support of advanced string filtering algorithms such as the bloom filter [31], which is already being deployed in most application level firewalls. We note that since WebSSARI also performs signature-based filtering to sanitize untrustworthy data, it is also subject to this problem.

3.1.2. Testing—how to reduce false negatives?

WSSs available to date suffer from the high rate of false negatives due to two main reasons. Firstly, bypassing form validation procedures is difficult. Some WSSs, such as VeriWeb [12] and AppScan [94], adopt a profile-based solution that requires administrators to manually supply valid values for every form field. WAVES incorporates techniques associated with hidden Web crawling to address this problem. However, even with such a mechanism in place, enumerating all execution paths is difficult. For example, for many websites, a majority of DEPs will not be identified if the webcrawler does not complete a login form correctly. Even if the webcrawler is capable of recognizing a login form, the administrator must manually supply proper values (i.e., a pair of valid username and password). These suggest that manual efforts are unavoidable in order to reduce false negatives. The second reason is that current WSSs use malicious patterns to detect SQL injection vulnerabilities. They submit malicious patterns to DEPs and observe their output. A majority of these malicious patterns are designed to make the backend database of vulnerable Web applications output error messages. Such error messages are then delivered by a Web application as parts of its output and observed by the WSSs. However, many Web applications today suppress such error messages, and therefore subject current WSS testing methodology to a high rate of false negatives.

3.1.3. Verification—Web languages are hard to verify

WebSSARI incorporates a compile-time verification algorithm that statically approximates runtime state. Such approximation is harder for weakly-typed languages, and for languages that support features such as pointer aliasing, function pointers, and object-oriented programming. These features often cause the number of states of a verifier to grow exponentially, making the task of verifying larger programs nearly impossible. Popular languages used for Web application development, such as PHP and Perl, not only support all the above features, but are also *scripting languages*. Scripting languages are not compiled into executables but executed directly by interpreters. Therefore, they have a much looser construct and support *dynamic evaluation*—that is, they can generate code on the fly and have the interpreter execute them. In other words, they can programmatically interact with the underlying interpreter at runtime. All these features make it very difficult for runtime state approximation. Before the Web, complex software were seldom developed using scripting languages, and therefore not much efforts have been made to study the verification of scripting languages. However, today's Web applications are large and complex, but a majority of them are developed using scripting languages. To successfully verify these applications, techniques must be developed to handle features (such as dynamic evaluation) unique to scripting languages.

4. REFERENCES

[1] Allen, F. E, Cocke, J. "A Program Data Flow Analysis Procedure." Communications of the ACM, 19(3):13147, March 1976.

[2] Andrews, G. R., Reitman, R. P. "An Axiomatic Approach to Information Flow in Programs." ACM Transactions on Programming Languages and Systems, 2(1), 56-76, 1980.

[3] Ashcraft, K., Engler, D. "Using Programmer-Written Compiler Extensions to Catch Security Holes." In Proceedings of the 2002 IEEE Symposium on Security and Privacy, pages 131-147, Oakland, California, 2002.

[4] Augustin, L., Bressler, D., Smith, G. "Accelerating Software Development through Collaboration." In Proceedings of the 24th International Conference on Software Engineering, pages 559-563, Orlando, Florida, May 19-25, 2002.

[5] Auronen, L. “Tool-Based Approach to Assessing Web Application Security.” Helsinki University of Technology, Nov 2002.

[6] Ball, T., Rajamani, S. K., "Automatically Validating Temporal Safety Properties of Interfaces." In Proceedings of the 8th International SPIN Workshop on Model Checking of Software, pages 103-122, volume LNCS 2057, Toronto, Canada, May 19-21, 2001. Springer-Verlag.

[7] Balzer, R., “Assuring the safety of opening email attachments.” In: DARPA Information Survivability Conference & Exposition II, 2, 257-262, 2001.

[8] Banatre, J. P., Bryce, C., Le Metayer, D. "Compile-time Detection of Information Flow in Sequential Programs." In Proceedings of the Third European Symposium on Research in Computer Security, pages 55-73, volume LNCS 875, Brighton, UK, Nov 1994. Springer-Verlag.

[9] Banerjee, A., Naumann, D.A. "Secure Information Flow and Pointer confinement in a Java-Like Language." In: Proceedings of the 15th Computer Security Foundations Workshop, pages 239-253, Nova Scotia, Canada, 2002.

[10] Barth, J. M. "A Practical Interprocedural Data Flow Analysis Algorithm." Communications of the ACM, 21(9):724-736, 1978.

[11] Bell, D. E., La Padula, L. J. "Secure Computer System: Unified Exposition and Multics Interpretation." Tech Rep. ESD-TR-75-306, MITRE Corporation, 1976.

[12] Benedikt M., Freire J., Godefroid P., “VeriWeb: Automatically Testing Dynamic Web Sites.” In: Proceedings of the 11th International Conference on the World Wide Web (Honolulu, Hawaii, May 2002).

[13] Bergman, M. K. “The Deep Web: Surfacing Hidden Value.” Deep Content Whitepaper, 2001.

[14] Biba, K. J. "Integrity Considerations for Secure Computer Systems." Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Massachusetts, Apr 1977.

[15] Bishop, M., Dilger, M. “Checking for Race Conditions in File Accesses.” Computing Systems, 9(2):131-152, Spring 1996.

[16] Bobbitt, M. "Bulletproof Web Security." Network Security Magazine, TechTarget Storage Media, May 2002. http://infosecuritymag.techtarget.com/2002/may/bulletproof.shtml

[17] Bowman, C. M., Danzig, P., Hardy, D., Manber, U., Schwartz, M., Wessels, D. “Harvest: A Scalable, Customizable Discovery and Access System." In: Technical Report CU-CS-732-94.”, Department of Computer Science, University of Colorado, Boulder, 1995.

[18] CERT. “CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests.” http://www.cgisecurity.com/articles/xss-faq.shtml

[19] Chen, H., Wagner, D. “MOPS: an Infrastructure for Examining Security Properties of Software.” In: ACM conference on computer and communication security (Washington, D.C., Nov 2002).

[20] Cho, J., Garcia-Molina, H. “Parallel Crawlers.” In: Proceedings of the 11th International Conference on the World Wide Web (Honolulu, Hawaii, May 2002), 124-135.

[21] Cousot, P., Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints." Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pages 238-252, 1977.

[22] Cowan, C., D. Maier, C. Pu, Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H. “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks.” In Proceedings of the 7th USENIX Security Conference, pages 63--78, San Antonio, Texas, Jan 1998.

[23] Cowan, C. "Software Security for Open-Source Systems." IEEE Security and Privacy, 1(1):38-45, 2003.

[24] Curphey, M., Endler, D., Hau, W., Taylor, S., Smith, T., Russell, A., McKenna, G., Parke, R., McLaughlin, K., Tranter, N., Klien, A., Groves, D., By-Gad, I., Huseby, S., Eizner, M., McNamara, R. “A Guide to Building Secure Web Applications.” The Open Web Application Security Project, v.1.1.1, Sep 2002.

[25] Darvas, A., Hähnle, R., Sands, D. "A Theorem Proving Approach to Analysis of Secure Information Flow." In Proceedings of the Workshop on Issues in the Theory of Security, Warsaw, Poland, Apr 5-6, 2003.

[26] Das, M., Lerner, S., Seigle, M. "ESP: Path-Sensitive Program Verification in Polynomial Time." In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 57-68, Berlin, Germany, 2002.

[27] DeKok, A. "PScan: A Limited Problem Scanner for C Source Files." http://www.striker.ottawa.on.ca/~aland/pscan/

[28] DeLine, R. Fahndrich, M. "Enforcing High-Level Protocols in Low-Level Software." In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pages 59-69, Snowbird, Utah, 2001.

[29] Denning, D. E. "A Lattice Model of Secure Information Flow." Communications of the ACM, 19(5):236-243, 1976.

[30] DeRemer, F. "Simple LR(k) Grammars." Communications of the ACM, 14(7):453-460, 1971.

[31] Dharmapurikar, S., Krishnamurthy, P., Sproull, T., and Lockwood, J. "Deep Packet Inspection Using Parallel Bloom Filters." In Proceedings of the 11th Symposium on High Performance Interconnects, pages 44-51, Stanford, California, 2003.

[32] DHTML Central. HierMenus.

[33] http://www.webreference.com/dhtml/hiermenus/

[34] Di Lucca, G.A.; Di Penta, M.; Antoniol, G.; Casazza, G. "An approach for reverse engineering of web-based applications." In: Proceedings of the Eighth Working Conference on Reverse Engineering (Stuttgart, Germany, Oct 2001), 231-240.

[35] Di Lucca, G.A., Fasolino, A.R., Pace, F., Tramontana, P., De Carlini, U. "WARE: a tool for the reverse engineering of web applications." In: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (Budapest, Hungary, Mar 2002), 241- 250.

[36] Doh, K. G., Shin, S. C. "Detection of Information Leak by Data Flow Analysis." ACM SIGPLAN Notices, 37(8):66-71, 2002.

[37] Evans D., Larochelle, D. "Improving Security Using Extensible Lightweight Static Analysis." In: IEEE Software, Jan 2002.

[38] Federal Trade Commission. "Security Check: Reducing Risks to your Computer Systems." 2003. http://www.ftc.gov/bcp/conline/pubs/buspubs/security.htm

[39] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. "Extended Static Checking for Java." In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 234-245, volume 37(5) of ACM SIGPLAN Notices, Berlin, Germany, Jun 2002.

[40] Foster, J. S., Fähndrich, M., Aiken, A. "A Theory of Type Qualifiers." In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pages 192--203, volume 34(5) of ACM SIGPLAN Notices, Atlanta, Georgia, May 1-4, 1999.

[41] Foster, J., Terauchi, T., Aiken, A. "Flow-Sensitive Type Qualifiers." In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pages 1-12, Berlin, Jun 2002.

[42] Gagnon, E. M., Hendren, L. J. "SableCC, an Object-oOiented Compiler Framework." In Proceedings of the 1998 Conference on Technology of Object-Oriented Languages and Systems (TOOLS-98), pages 140-154, Santa Barbara, California, Aug 3-7, 1998.

[43] Goguen, J. A., Meseguer, J. "Security Policies and Security Models." In Proceedings of the IEEE Symposium on Security and Privacy, pages 11-20, Oakland, California, Apr 1982.

[44] Graham, S., Wegman, M. "A Fast and Usually Linear Algorithm for Global Flow Analysis." Journal of the ACM, 23(1):172-202, Janu 1976.

[45] Guyer, S. Z., Berger, E. D., Lin, C. "Detecting Errors with Configurable Whole-program Dataflow Analysis." Technical Report, UTCS TR-02-04, The University of Texas at Austin, 2002.

[46] Hallem, S., Chelf, B., Xie, Y., Engler, D. "A System and Language for Building System-Specific, Static Analyses." In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pages 69-82, Berlin, Germany, 2002.

[47] Hecht, M. S., Ullman, J. D. "Analysis of a Simple Algorithm For Global Flow Problems." In Conference Record of the First ACM Symposium on the Principles of Programming Languages, pages 207-217, Boston, Massachussets, 1973.

[48] Henglein, F. "Dynamic Typing." In Proceedings of the Fourth European Symposium on Programming, pages 233-253, volume LNCS 582, Rennes, France, Feb 1992. Springer-Verlag.

[49] Higgins, M., Ahmad, D., Arnold, C. L., Dunphy, B., Prosser, M., and Weafer, V., "Symantec Internet Security Threat Report—Attack Trends for Q3 and Q4 2002," Symantec, Feb 2003.

[50] Holzmann, G. J. "The Logic of Bugs." In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 81-87, Charleston, South Carolina, 2002.

[51] Huang, Y. W., Huang, S. K., Lin, T. P., Tsai, C. H. "Web Application Security Assessment by Fault Injection and Behavior Monitoring." In Proceedings of the Twelfth International World Wide Web Conference, 148-159, Budapest, Hungary, May 21-25, 2003.

[52] Huang, Y. W., Tsai, C. H., Lee, D. T., Kuo, S. Y. "Non-Detrimental Web Application Security Auditing." In *Proceedings of the Fifteenth IEEE International Symposium on Software Reliability Engineering (ISSRE2004)*, Nov 2-5, Rennes and Saint-Malo, France, 2004.

[53] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., Kuo, S. Y. "Securing Web Application Code by Static Analysis and Runtime Protection." In *Proceedings of the Thirteenth International World Wide Web Conference (WWW2004)*, pages 40-52, New York, May 17-22, 2004.

[54] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., Kuo, S. Y. "Verifying Web Applications Using Bounded Model Checking." In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN2004)*, pages 199-208, Florence, Italy, Jun 28-Jul 1, 2004.

[55] Hughes, F. "PHP: Most Popular Server-Side Web Scripting Technology." LWN.net. http://lwn.net/Articles/1433/

[56] Ipeirotis P., Gravano L., "Distributed Search over the Hidden Web: Hierarchical Database Sampling and Selection." In: The 28th International Conference on Very Large Databases (Hong Kong, China, Aug 2002), 394-405.

[57] Jensen, T., Le Metayer, D., Thorn, T. "Verification of Control Flow Based Security Properties." In Proceedings of the 20th IEEE Symposium on Security and Privacy, pages 89-103, IEEE Computer Society, New York, USA, 1999.

[58] Joshi, R., Leino, K. M. "A Semantic Approach to Secure Information Flow." Science of Computer Programming, 37(1-3):113-138, 2000.

[59] Joshi, J., Aref, W., Ghafoor, A., Spafford, E. "Security Models for Web-Based Applications." Communications of the ACM, 44(2), 38-44, Feb 2001.

[60] Kavado, Inc. "InterDo Version 3.0." Kavado Whitepaper, 2003.

[61] Ko, C., Fraser, T., Badger, L., Kilpatrick, D. "Detecting and Countering System Intrusions Using Software Wrappers." In: Proceedings of the 9th USENIX Security Symposium (Denver, Colorado, Aug 2000).

[62] Krishnamurthy, A. "Hotmail, Yahoo in the run to rectify filter flaw." TechTree.com, March 24, 2004. http://www.techtree.com/techtree/jsp/showstory.jsp?storyid=5038

[63] Larochelle, D., Evans, D. "Statically Detecting Likely Buffer Overflow Vulnerabilites." In Proceedings of the 10th USENIX Security Symposium, Washington, D.C., Aug 2001.

[64] Liddle, S., Embley, D., Scott, D., Yau, S.H., "Extracting Data Behind Web Forms." In: Proceedings of the Workshop on Conceptual Modeling Approaches for e-Business (Tampere, Finland, Oct 2002).

[65] Manber, U., Smith, M., Gopal B., "WebGlimpse—Combining Browsing and Searching." In: Proceedings of the USENIX 1997 Annual Technical Conference (Anaheim, California, Jan, 1997).

[66] Mandre, I. "PHP 4 Grammar for SableCC 3 Complete with Transformations." Indrek's SableCC Page, 2003. http://www.mare.ee/indrek/sablecc/

[67] Maor O., Shulman, A., "SQL Injection Signatures Evasion." Imperva, Inc., Apr 2004.

[68] Meier, J.D., Mackman, A., Vasireddy, S. Dunner, M., Escamilla, R., Murukan, A. "Inproving Web Application Security—Threats and Countermeasures." Microsoft Corporation, 2003.

[69] Microsoft. "Scriptlet Security." Getting Started with Scriptlets, MSDN Library, 1997 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnindhtm/html/instantdhtmlscriptlets.asp

[70] Microsoft. "Visual C++ Compiler Options: /GS (Buffer Security Check)." MSDN Library, 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/vclrfGSBufferSecurity.asp

[71] Miller, R. C., Bharat, K. “SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers.” In: Proceedings of the 7th International World Wide Web Conference (Brisbane, Australia, April 1998), 119-130.

[72] Mizuno, M., Schmidt, D. A. "A Security Flow Control Algorithm and Its Denotational Semantics Correctness Proof." Formal Aspects of Computing, 4(6A):727-754, 1992.

[73] Morrisett, G., Walker, D., Crary, K., Glew, N. "From System F to Typed Assembly Language." ACM Transactions on Programming Languages and Systems, 21(3):528-569, May 1999.

[74] Mozilla.org. “Mozilla Layout Engine.” http://www.mozilla.org/newlayout/

[75] Myers, A. C. "JFlow: Practical Mostly-Static Information Flow Control." In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 228-241, San Antonio, Texas, 1999.

[76] Necula, G. C. “Proof-Carrying Code.” In Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 106-119, Paris, France, Jan 1997.

[77] Necula, G. C., McPeak, S., Weimer, W. “CCured: Type-Safe Retrofitting of Legacy Code.” In Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 128-139, Portland, Oregon, 2002.

[78] Netscape. “JavaScript Security in Communicator 4.x.”

http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm#1023448

[79] Neumann, P. G. “Risks to the Public in Computers and Related Systems.” *ACM SIGSOFT Software Engineering Notes*, 25(3), p.15-23, 2000.

[80] Ohmaki, K. “Open Source Software Research Activities in AIST towards Secure Open Systems.” *In Proc. 7th IEEE Int’l Symp. High Assurance Systems Engineering (HASE'02)*, p.37, Tokyo, Japan, Oct 23-25, 2002.

[81] Orbaek, P. "Can You Trust Your Data?" In Proceedings of the 1995 TAPSOFT/FASE Conference, pages 575-590, volume LNCS 915, Aarhus, Denmark, May 1995. Springer-Verlag.

[82] OWASP. "The Ten Most Critical Web Application Security Vulnerabilities." OWASP Whitepaper, version 1.0, 2003.

[83] Park, J. S., Sandhu, R. "Role-Based Access Control on the Web." ACM Transactions on Information and System Security 4(1):37-71, 2001.

[84] Pottier, F., Simonet, V. "Information Flow Inference for ML." ACM Transactions on Programming Languages and Systems, 25(1):117-158, 2003.

[85] Raghavan, S., Garcia-Molina, H. “Crawling the Hidden Web.” In: Proceedings of the 27th VLDB Conference (Roma, Italy, Sep 2001), 129-138.

[86] Raghavan, S., Garcia-Molina, H. “Crawling the Hidden Web.” In: Technical Report 2000-36, Database Group, Computer Science Department, Stanford (Nov 2000).

[87] Rapps, S., Weyuker, E. J. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, SE-11, p.367-375, 1985.

[88] Ricca, F., Tonella, P. “Analysis and Testing of Web Applications.” In: Proceedings of the 23rd IEEE International Conference on Software Engineering (Toronto, Ontario, Canada, May 2001), 25 –34.

[89] Ricca, F., Tonella, P., Baxter, I. D. “Restructuring Web Applications via Transformation Rules.” Information and Software Technology, 44(13), 811-825, Oct 2002.

[90] Ricca, F., Tonella, P. “Understanding and Restructuring Web Sites with ReWeb.” IEEE Multimedia, 8(2), 40-51, Apr 2001.

[91] Ricca, F., Tonella, P. “Web Application Slicing.” In: Proceedings of the IEEE International Conference on Software Maintenance (Florence, Italy, Nov 2001), 148-157.

[92] Ricca, F., Tonella, P. “Web Site Analysis: Structure and Evolution.” In: Proceedings of the IEEE International Conference on Software Maintenance (San Jose, California, Oct 2000), 76-86.

[93] Sabelfeld, A., Myers, A. C. “Language-Based Information-Flow Security.” IEEE Journal on Selected Areas in Communications, 21(1):5-19, 2003.

[94] Sanctum Inc. “Web Application Security Testing – AppScan 3.5.” http://www.sanctuminc.com

[95] Sanctum Inc. “AppShield 4.0 Whitepaper.” 2002. http://www.sanctuminc.com

[96] Sandhu, R. S. "Lattice-Based Access Control Models." IEEE Computer, 26(11):9-19, 1993.

[97] Schneider, F. B. "Enforceable Security Policies." ACM Transactions on Information and System Security, 3(1):30-50, Feb 2000.

[98] Scott, D., Sharp, R. “Abstracting Application-Level Web Security.” In: The 11th International Conference on the World Wide Web (Honolulu, Hawaii, May 2002), 396-407.

[99] Scott, D., Sharp, R. “Developing Secure Web Applications.” IEEE Internet Computing, 6(6), 38-45, Nov 2,002.

[100] Sekar, R., Uppuluri, P., “Synthesizing Fast Intrusion Detection/Prevention Systems from High-Level Specifications.” In: USENIX Security Symposium, 1999.

[101] Sebastien@ailleret.com. “Larbin – A Multi-Purpose Web Crawler.” http://larbin.sourceforge.net/index-eng.html

[102] Secure Software, Inc. “RATS—Rough Auditing Tool for Security.” http://www.securesoftware.com/

[103] Shankar, U., Talwar, K., Foster, J. S., Wagner, D. “Detecting Format String Vulnerabilities with Type Qualifiers.” In Proceedings of the 10th USENIX Security Symposium, pages 201-220, Washington DC, Aug 2002.

[104] SPI Dynamics. “Web Application Security Assessment.” SPI Dynamics Whitepaper, 2003.

[105] Stiennon, R., "Magic Quadrant for Enterprise Firewalls, 1H03." Research Note. M-20-0110, Gartner, Inc., 2003.

[106] Strom, R. E., Yemini, S. A. "Typestate: A Programming Language Concept for Enhancing Software Reliability." IEEE Transactions on Software Engineering, 12(1):157-171, Jan 1986.

[107] Tennyson Maxwell Information Systems, Inc. “Teleport Webspiders.” http://www.tenmax.com/teleport/home.htm

[108] Varghese, S. “Microsoft patches critical Hotmail hole.” TheAge.com, March 24, 2004. http://www.theage.com.au/articles/2004/03/24/1079939690076.html

[109] Visa U.S.A. "Cardholder Information Security Program (CISP) Security Audit Procedures and Reporting as of 8/8/2003." Version 2.2, 2003.

[110] Viega, J., Bloch, J., Kohno, T., McGraw, G. "ITS4: a static vulnerability scanner for C and C++ code." In The 16th Annual Computer Security Applications Conference, New Orleans, Louisiana, Dec 11-15, 2000.

[111] Volpano, D., Smith, G., Irvine, C. “A Sound Type System For Secure Flow Analysis." Journal of Computer Security, 4(3):167-187, 1996.

[112] Wagner, D., Foster, J. S., Brewer, E. A., Aiken, A. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities." In Proceedings of the 7th Network and Distributed System Security Symposium, pages 3-17, San Diego, California, Feb 2000.

[113] Wall, L., Christiansen, T., Schwartz, R. L. "Programming Perl." O'Reilly and Associates, 3rd edition, July 2000.

[114] Walker, D. "A Type System for Expressive Security Policies." In Proceedings of the 27th Symposium on Principles of Programming Languages, pages 254-267, ACM Press, Boston, Massachusetts, Jan 2000.

[115] Watts, G. "PHPXref: PHP Cross Referencing Documentation Generator." Sep 2003. http://phpxref.sourceforge.net/

[116] Wheeler, D. A. “FlawFinder.” http://www.dwheeler.com/flawfinder/

[117] Wright, A. K, Cartwright, R. “A Practical Soft Type System for Scheme.” ACM Transactions on Programming Languages and Systems, 19(1):87-152, Jan 1999