Securing Web Application Code by Static Analysis and Runtime Protection

Yao-Wen Huang[+*], Fang Yu[*], Christian Hang[#], Chung-Hung Tsai[+], D. T. Lee[+*], Sy-Yen Kuo[+]
{ywhuang, yuf, dtlee}@iis.sinica.edu.tw, christian.hang@web.de, sykuo@cc.ee.ntu.edu.tw

[+]Department of Electrical Engineering, National Taiwan University Taipei 106, Taiwan.

[*]Institute of Information Science, Academia Sinica Taipei 115, Taiwan

[#] Department of Computer Science, RWTH Aachen Aachen, Germany

ABSTRACT
Security remains a major roadblock to universal acceptance of the Web for many kinds of transactions, especially since the recent sharp increase in remotely exploitable vulnerabilities has been attributed to Web application bugs. Many verification tools are discovering previously unknown vulnerabilities in legacy C programs, raising hopes that the same success can be achieved with Web applications. In this paper, we describe a sound and holistic approach to ensuring Web application security. Viewing Web application vulnerabilities as a secure information flow problem, we created a lattice-based static analysis algorithm derived from type systems and typestate, and addressed its soundness. 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. We also created a tool named *WebSSARI* (Web application Security by Static Analysis and Runtime Inspection) to test our algorithm, and used it to verify 230 open-source Web application projects on SourceForge.net, which were selected to represent projects of different maturity, popularity, and scale. 69 contained vulnerabilities and their developers were notified. 38 projects acknowledged our findings and stated their plans to provide patches. Our statistics also show that static analysis reduced potential runtime overhead by 98.4%.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software / Program Verification – *class invariants, formal methods;* D.4.6 [**Operating Systems**]: Security and Protection – *information flow controls, correctness proofs, formal methods;* K.6.5 [**Computing Milieux**]: Security and Protection – *invasive software, unauthorized access.*

General Terms
Security, Verification.

Keywords
Web application security, security vulnerabilities, program security, verification, type systems, information flow, noninterference.

Copyright is held by the author/owner(s).
WWW 2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

1. INTRODUCTION
As more and more services are provided via the World Wide Web, efforts from both academia and industry are striving to create technologies and standards that meet the sophisticated requirements of today's Web applications and users. In many situations, security remains a major roadblock to universal acceptance of the Web for all kinds of transactions. According to a Symantec report released earlier this year, over the preceding 12 months in 2002, there was an 81.5% increase in documented vulnerabilities, with the majority associated with a handful of very severe vulnerabilities [36]. The report's authors suggested that the driving force behind this trend is "the rapid development and deployment of remotely exploitable Web applications." They reported that the total number of Web application vulnerabilities discovered in 2002 was 178% higher than in 2001, that 95 percent of these were remotely exploitable, and that 99 percent were considered highly or moderately severe.

Scott and Sharp [62] [63] 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. Current technologies such as anti-virus software programs and network firewalls offer comparatively secure protection at the host and network levels, but not at the application level [17]. However, when network and host-level entry points are relatively secure, the public interfaces of Web applications become the focus of attacks [46] [17].

The recognition of this problem is reflected by the recent burst of efforts that aim to improve Web application security via numerous different approaches. Scott and Sharp proposed the use of a gateway that filters invalid and malicious inputs at the application level; Sanctum's AppShield [58], Kavado's InterDo [43], and a number of commercial products now offer similar strategies. Most of the leading firewall vendors are also using deep packet inspection [24] technologies in their attempts to filter application-level traffic. According to a recent Gartner report [67], 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 two drawbacks: they require careful configuration [12], and they only offer Web application protection (that is, they don't identify errors). Huang et al. [38] designed a Web application security assessment framework that offers black-boxed testing for identifying Web application vulnerabilities. However, testing processes cannot guarantee identification of all bugs, and they cannot provide

immediate security for Web applications in the same manner that Scott and Sharp's solution can. In the present project, we tried to create an approach that simultaneously provides immediate security for Web applications and identifies all vulnerabilities within their code.

As we will discuss in the next section, a major challenge associated with Web applications is that their most critical vulnerabilities are often the results of insecure information flow, against which neither encryption nor traditional Web access control models [55] offer any protection [57]. Sabelfeld and Myers [57] recently published a comprehensive survey on language-based techniques for specifying and enforcing information-flow policies. Among them, sound type systems [70] based on the lattice model of Denning [22] appear most promising. Banerjee and Naumann [7] proposed such a system for a Java-like language, and Pottier and Simonet [56] proposed one for ML. Myers [50] 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) [40]. Using a type qualifier theory [27], Shankar et al. [65] detected insecure information flow within legacy code with little additional annotation. Using metacompilation-based checkers [33], 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 [15] 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 [47].

Our project goal is to use a mix of static and runtime features to establish a holistic and practical approach to ensuring Web application security. To achieve this, we have created a tool that a) statically verifies existing Web application code without any additional annotation effort; and b) after verification, automatically secures potentially vulnerable sections of the code.

This paper has the following contributions:

1. We have shown that most Web application security problems arise from data integrity violations caused by insecure information flow, and that mechanisms are needed to express and enforce *noninterference* policies [30].
2. For specifying and verifying noninterference policies, we have proposed a type system based on Denning's axioms [22] and Strom's *typestate* [68]. The system's advantages are twofold: first, it captures information-flow semantics more precisely than static systems, resulting in lower false positive rates; second, it requires no annotation effort on the part of programmers.
3. Our proposed system acts as an extension to a language's existing type system. We have implemented *WebSSARI* (Web application Security by Static Analysis and Runtime Inspection) as a framework for extending existing script languages with our system. Currently, WebSSARI supports PHP—the most widely used Web application programming language [40]. Given the corresponding grammar, WebSSARI can also support other languages used for Web application programming.
4. WebSSARI automatically inserts runtime guards in potentially insecure sections of code, meaning that a piece of PHP code will be secured immediately after WebSSARI processing even in the absence of programmer intervention. Induced overhead is low because the number of insertions is reduced to a minimum when information gathered from static analysis is utilized. Users can add annotations to further reduce this number, possibly to zero.
5. We have implemented our algorithm into WebSSARI. We used it to verify 230 open-source Web application projects on SourceForge.net, which were selected to represent projects of different maturity, popularity, and scale. 69 projects, among which many were widely-used, contained vulnerabilities. Numerous discovered vulnerabilities allowed remote attackers to completely compromise machines running the software. Upon notification, developers of 38 projects acknowledged our findings and stated their plans to provide patches. Our statistics also show that static analysis reduced potential runtime overhead by 98.4%.

To the best of our knowledge, such a tool has never been provided for and experimented with real-world Web application code.

2. WEB APPLICATION VULNERABILITIES

In this section we will give several brief examples of Web application vulnerabilities. Since we will only provide brief descriptions of the most widely exploited vulnerability—script injection—readers are referred to Scott and Sharp [62] [63], Curphey et al. [17], Curphey et al. [54] [17], and Meier et al. [46] for more details.

2.1 Cross-Site Scripting (XSS)

Cross-site scripting (XSS) is perhaps the most common Web application vulnerability. Figure 1 gives an example of an XSS bug we identified in *SquirrelMail*, a popular Web-based e-mail application.

```
$month=$_GET['month']; $year=$_GET['year'];
$day=$_GET['day'];
echo "<a href=\"day.php?year=$year&amp;"
echo "month=$month&amp;day=$day\">";
```

Figure 1. An XSS vulnerability found in *SquirrelMail*.

Values for the variables $month, $day, and $year 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/event_delete.php?year=><script>malicious_script();</script>

Attackers must find ways to make victims open this URL. One strategy is to send an e-mail containing javascript that secretly launches a hidden browser window to open this URL. Another is to embed the same javascript inside a Web page; 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.

```
<a href= "day.php?year=><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 [62] 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; 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.

2.2 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. The example below is based on a vulnerability we discovered in *ILIAS Open Source*, a popular Web-based learning management system.

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

Figure 3. A simplified SQL injection vulnerability found in *ILIAS Open Source*.

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

```
');DROP TABLE ('users
```

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

```
INSERT INTO tracking_temp VALUES('');
DROP TABLE ('users');
```

Table "users" will be dropped 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. During our experimentation with WebSSARI, we found that developers who acknowledged that variables from HTTP requests should not be trusted tend to forget that the same holds true for the referrer field, user cookies, and other types of information collected from HTTP requests.

2.3 General Script Injection

General script injection vulnerabilities are considered the most severe of the three types discussed in this paper. 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 we found in *eGroupWare*, a widely-adopted Web-based groupware suite sponsored by Toshiba. The HTTP request variable "csvfile" is used as an argument to call fopen(), which allows arbitrary files to be opened. A subsequent code section delivers the opened file to the HTTP client, allowing attackers to download arbitrary files.

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

Figure 4. A general script injection bug found in *eGroupWare*.

A more severe example of this vulnerability type—a bug we found in the *PHP Surveyor* online survey management system—is shown in Figure 5.

```
exec("htpasswd.exe -b .htpasswd".
"{$_POST['user']} {$_POST['pass']}");
```

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

The intent for this code is to allow survey administrators to change user passwords for system access. However, since the "user" and "pass" variables are untrusted, the code permits the execution of arbitrary system commands. For instance, if a malicious survey administrator sends an HTTP request with: `user="; NET USER foo /ADD"` and `pass=""`, the actual command becomes:

```
htpasswd.exe -b .htpasswd; NET USER FOO /ADD
```

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

2.4 Modeling Web Application Vulnerabilities

The primary objectives of information security systems are to protect confidentiality, integrity, and availability [60]. From our examples, 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. Untrusted data is used to construct trusted output without sanitization, resulting in data integrity violations. This leads to escalation of access rights, which then results in compromises in availability and confidentiality. There is clearly a need for a mechanism that specifies and enforces secure information flow policies within Web application programs.

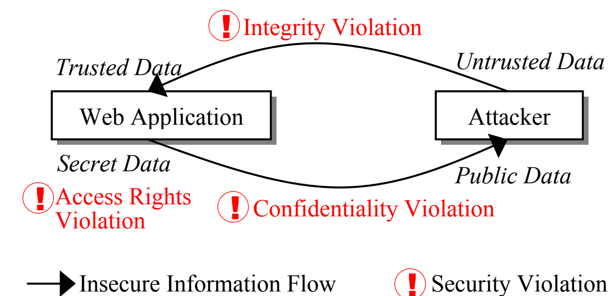

Figure 6. A model of common Web application vulnerabilities.

3. INFORMATION FLOW SECURITY

Type systems have proven useful for specifying and checking program safety properties. By means of programmer-supplied annotations, both proof-carrying codes (PCC) [51] and typed assembly languages (TAL) [49] 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 (PHP) and tried to avoid additional annotations.

Many previous security verification efforts have focused on temporal safety properties related to control flow. Schneider [61] proposed formalizing security properties using *security automata*, which define the legal sequences of program actions. Walker [73] proposed a TAL extension which uses security policies expressed in Schneider's automata to derive its type system. Jensen et al. [41] 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 [13] 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 [5] adopted a similar approach for their SLAM project and successfully applied it to Windows XP device drivers.

3.1 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 [9]. 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; Biba [10] is credited with adding the concept of integrity.

Denning [22] established a lattice model for analyzing secure information flow in imperative programming languages based on a program abstraction (similar to Cousot and Cousot's [14] *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 [48]). Orbaek [53] proposed a similar treatment, but addressed the secure information flow problem in terms of data integrity instead of confidentiality. Volpano et al. [70] argued that both works proved soundness with respect to some instrumented semantics whose validity was open to question; 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 et al. 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 [30] 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 [56] and Java [50] [7]. Based on Foster's theory of type qualifiers [27], Shankar et al. [65] 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 that they were previously unaware of.

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 rate, which often makes them impractical for real-world use. Regardless of whether security classes are assigned through manual annotations or through inference rules, they are statically bound to program variables in conventional type systems. 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 [50], each program variable is associated with a fixed security label. 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 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 variable labels.

3.2 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 [25] *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 label 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 more precise approximations of the run-time information flow. 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. Banatre et al. [6] have offered a comparable approach plus a proof checking method that resembles dataflow analysis techniques associated with optimizing compilers. Joshi and Leino [42] examined various logical forms for representing information flow semantics, leading to a characterization containing Hoare triples. Darvas et al. [18] 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 involving flow-sensitive analysis techniques used by optimizing compilers has have been extensively researched starting from the early works of Allen and Cocke [1] and followed by the works of Hecht and Ullman [34], Graham and Wegman [31], Barth [8], 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 scalability. ESP, the verification

tool recently developed by Das et al. [19], is representative of this approach; the contribution is distinctive because ESP allows for flow-sensitive verification that scales to large programs. It is based on the assumption that most program branches do not affect the information flow property that is being checked. Unlike ESP, Guyer et al.'s [32] approach has a specific security focus. It used the flow-sensitive, context-sensitive, interprocedural data flow analysis framework provided by their Broadway optimizing compiler to check for format string vulnerabilities of real C programs.

3.3 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. [65] 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* [68], 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 that type-based information flow systems suffer. Inspired by typestate, DeLine and Fahndrich [21] 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. [28] 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 [19], 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 incorporates 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.

3.4 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 [11], 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* [33] 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, interprocedural data flow checking framework that requires no additional annotations. In contrast, Flanagan et al.'s ESC/Java [26] (designed to check the correctness of Java programs) requires additional annotations from programmers.

Most efforts to develop checkers have resulted in publicly available tools [16], including BOON by Wagner et al. [71], RATS by Secure Software [64], FlawFinder by Wheeler [75], PScan by DeKok [20], Splint by Larochelle and Evans [44], and ITS4 by Viega et al. [69]. 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.

3.5 A Comparison

Most previous type-based static verification methods are provided as extensions to existing languages (e.g., Pottier and Simonet [56], Banerjee and Naumann [7], and Myers [50]) and designed to support secure program development, while our algorithm attempts to verify existing code in the absence of user intervention. Checkers (e.g., MC [3], RATS [64], and ITS4 [69]) also perform dataflow analysis without additional annotations, but their analyses are considered unsound. Another difference is that WebSSARI ensures security by inserting runtime guards, while the other tools are limited to providing verification.

	Focus	App	Snd	Anno	Lang	Dec
WebSSARI	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.	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 7. A comparison with related work.

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; 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 most Web programming languages (e.g., PHP, ASP, and Perl) provide 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. [65]), and achieved a further reduction in the number of class 2 errors.

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 7.

3.6 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" [72], which ensures system integrity by tracking tainted data submitted by the user at runtime. Similarly, Myers [50] also leaves some JIF security label 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 were undeterminable at compile-time. These kinds of dynamic checks are extremely expensive, resulting in the creation of such static optimization techniques as dynamic typing [35] and soft typing [77] 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 et al.'s *CCured* [52]. Though not specifically focused on security, this 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. When inserted at the proper locations, their execution time cannot be considered real overhead because the action is a necessary security check; WebSSARI will have simply inserted lines of code omitted by a careless (or security-unaware) programmer.

3.7 Existing Web Application Security Mechanisms

Scott and Sharp [62] [63] used an application proxy to abstract Web application protection; the proxy validates user input, thus preventing untrustworthy input from entering Web applications. Commercial products such as AppShield [58] and InterDo [43] use a similar approach. However, even though it provides immediate assurance of Web application security, it requires the correct identification of and validation policy for each individual entry point to a Web application. As Bobbitt notes, this is a difficult security task that requires careful configuration by "highly technical, experienced individuals" [12]. Another limitation is that this approach protects Web applications at the deployment phase instead of trying to eliminate bugs during the development phase.

In light of these deficiencies, Huang et al. proposed an assessment framework for Web application security that they call WAVES (Web Application Vulnerability and Error Scanner) [38]. Their framework uses black-boxed testing to identify Web application vulnerabilities. Similar approaches are adopted by commercial projects such as *AppScan* [59], *WebInspect* [66], and *ScanDo* [43]. While this approach can be used to identify errors early in the development cycle, it does not provide immediate Web application protection as offered by Scott and Sharp's solution. Here we have tried to design an approach that retains the advantages of both while also eliminating their disadvantages.

4. VERIFICATION ALGORITHM

In PHP (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*; vulnerabilities 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.

4.1 Information Flow Model

To characterize data trust levels, we followed Denning's [22] 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 partially ordered by ≤, which is reflexive, transitive, and antisymmetric. For $\tau_1, \tau_2 \in T$,

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

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

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 needed to provide a method to express the trust levels of variables. Following Foster et al. [27] and Shankar et al. [65], 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 [27]. 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 functions 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 reduced overhead.

However, unlike Shankar et al. [65], we did not perform type inference (of security classes) in our attempt to eliminate user

annotation efforts. In conventional type-based secure information flow systems (e.g., JIF [50]), type inference 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 class throughout the entire program execution. As explained in Section 3.1, fixed variable classes induce false positives. To develop a type system in which variable classes can change and flow-sensitive properties can be considered, we maintain our *type environment* based on Strom and Yemini's [68] typestate.

A type environment $\Gamma : X \mapsto T$ is a mapping function that maps variables to security classes at a particular program point. For each variable $x \in \text{dom}(\Gamma)$, we denote the uniquely mapped type τ of x in Γ as $\Gamma(x)$. To approximate runtime type environment 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 x is mapped to $\Gamma(x)$ at a particular program point, then its corresponding execution time data object will have a class that is at most as restrictive as $\Gamma(x)$, regardless of which paths were taken to reach that point. Formally, for a set of type environments G, we denote $\Gamma = \oplus G$ as the most restrictive type environment, such that $\Gamma(x) = \sqcup_{\Gamma' \in G} \Gamma'(x)$. When verifying a program at a particular program point r, $\Gamma = \oplus G_r$, where G_r represents the set of all possible type environments, each corresponding to a unique execution-time path that could have been taken to reach r.

To illustrate this concept, we will use the widely-adopted tainted-untainted (T-U) lattice of security classes (e.g., by BOON [71], Ashcraft and Engler [3], and Shankar et al. [65]) shown in Figure 8. The T-U lattice has only two elements—untainted as its lower bound and tainted as its upper bound. Assume that variable t1 is tainted and that variables u1 and u2 are untainted. Since exec() requires an untainted argument, for line 2 of Figures 10 and 11 to typecheck requires that we know the static most restrictive class of x. In other words, we need to know the security class $\Gamma(x)$ 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 10, since x can be either tainted or untainted, $\Gamma(x) = \mathit{tainted} \sqcup \mathit{untainted}$; line 2 therefore triggers a violation. On the other hand, line 2 of Figure 11 typechecks.

Figure 8. Primitive lattice.

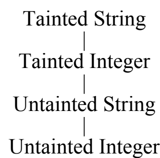

Figure 9. Type-aware lattice.

```
1: if (C) x = t1; else x = u1;
2: exec(x);
```

Figure 10. Example A.

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

Figure 11. Example B.

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 [68]. 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 9), 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 operator instead.

4.2 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 12. In that example, since `$_POST['index']` is tainted, \$i is tainted after line 1, and \$s is tainted after line 2. Line 3 therefore does not typecheck, since `echo()` requires untainted values for its argument.

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

Figure 12. 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 9 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. [27] [28] [65] [26]. In this paper 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.

4.3 Program Abstraction and Type Judgment

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

$$(\mathit{commands}) \quad c ::= c_1 ; c_2 \mid x := e \mid e \mid \text{if } e \text{ than } c_1 \text{ else } c_2$$
$$(\mathit{expression}) \quad e ::= x \mid n \mid e_1 \sim e_2 \mid f(x),$$

where x is a variable, n represents a constant value, ~ represents binary operators (e.g., +), $f(x)$ represents a function call. Commands that do not induce insecure flows are referred to as *valid* commands. To track the changes in security classes of variables and to check for program validity, we define *type judgments* and *judgment rules*. Denoted as $\Gamma \vdash C \to \Gamma'$, a type judgment specifies a type environment Γ in which the execution of a command C is valid, and becomes Γ' as a result of the execution. As stated in Section 4.1, preconditions of sensitive functions and postconditions of tainting and sanitization functions are defined in the prelude files. At call sites to sensitive functions,

$\text{SATISFY}(\Gamma, f, x)$ checks whether $\Gamma(x)$ satisfies function f's precondition. When verifying, we derive type judgments according to command sequences and raise an error when $\text{SATISFY}(\Gamma, f, x)$ fails. That is, given a program P and its initial type environment Γ_0 (usually mapping all variables to untainted), then the validity of P depends on whether we can derive the judgment $\Gamma_0 \vdash P \to \Gamma$ by following the judgment rules below.

1. Updating Rules:

(Pollution)

$$\frac{f \in \mathrm{T}}{\Gamma \vdash f(x) \to \Gamma[x \mapsto \mathit{tainted}]}$$

(Sanitation)

$$\frac{f \in \mathrm{S}}{\Gamma \vdash f(x) \to \Gamma[x \mapsto \mathit{untainted}]}$$

(Assignment)

$$\Gamma \vdash x := e \to \Gamma\left[x \mapsto \Gamma(e)\right]$$

(Restriction)

$$\frac{\Gamma \vdash C_1 \to \Gamma_1 \quad \Gamma \vdash C_2 \to \Gamma_2}{\Gamma \vdash \text{if } e \text{ then } C_1 \text{ else } C_2 \to \Gamma_1 \oplus \Gamma_2}$$

2. Checking Rule:

(Precondition)

$$\frac{f \in \mathrm{C},\ \text{SATISFY}(\Gamma, f, x)}{\Gamma \vdash f(x) \to \Gamma}$$

3. Concatenation Rule:

(Concatenation)

$$\frac{\Gamma \vdash C \to \Gamma'' \quad \Gamma'' \vdash C' \to \Gamma'}{\Gamma \vdash C; C' \to \Gamma'}$$

4. Mapping Rules:

$$\Gamma(n) = \mathit{untainted} \qquad \Gamma(e \sim e') = \Gamma(e) \sqcup \Gamma(e')$$

4.4 Soundness

At every program point, since we always derive the static most restrictive type environment, all variables are mapped to their most restrictive types among all execution paths to reach that point. This is an essential property that ensures the soundness of our algorithm. However, like many other popular languages used for Web development, PHP is a *scripting language* that supports *dynamic evaluation*—a feature unique to interpreted languages that allows for programmatic access to the interpreter. 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 (e.g., PHP's eval()), 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 [68]. 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.

5. SYSTEM IMPLEMENTATION

To test our approach, we developed a tool called *WebSSARI* that extends an existing script language with our proposed type qualifier system. An illustration of WebSSARI's system architecture is presented in Figure 13. 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). Based on the prelude files, the *verification engine* moves through the CFG and references the ST to generate a) type qualifiers for variables 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, and users can supply the prelude with their own routines.

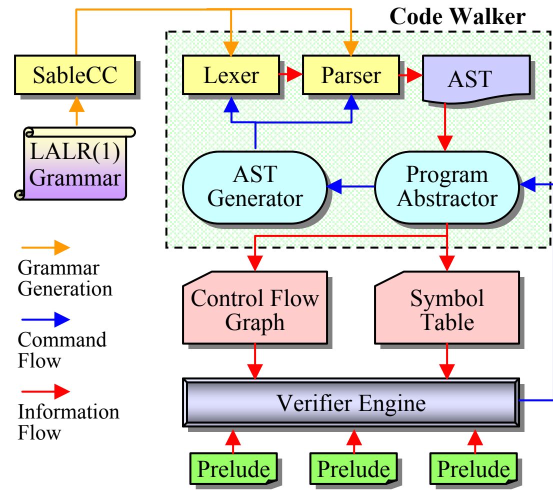

Figure 13. WebSSARI system architecture.

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; 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 sourcecode 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* [74] to generate cross referenced documentation of PHP source files. A screenshot of this GUI is presented in Figure 14.

In this project, we provided a code walker for PHP. We used Gagnon and Hendren's *SableCC* [29], an object-oriented compiler framework for Java. Similar to yacc and other compiler generators, SableCC accepts LALR(1) [23] 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). We used Mandre's [45] 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 following test involving 10,000 PHP files.

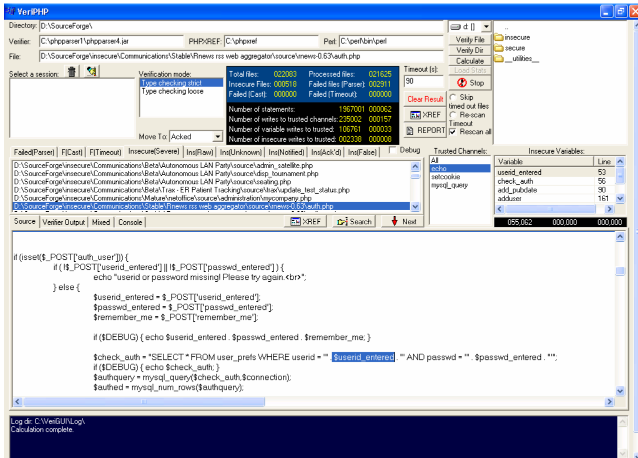

Figure 14. A screenshot of the WebSSARI GUI under Windows.

6. 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 programming 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 created 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 (Figure 15).

For each project it hosts, SourceForge.net assigns a "development status" (planning, pre-alpha, alpha, beta, stable, mature, and inactive) and an "activity rate" (indicating both development activity and popularity. We limited our selections to beta, stable, and mature products. Figure 16 presents the development status and activity rates of the 69 vulnerable projects, and Figure 17 presents distribution information for the 38 projects whose developers responded to our email notifications. We assumed that beta projects would be more vulnerable, but the data reflect the opposite—that is, stable stage projects were slightly more vulnerable than beta stage projects. These respective percentages were even higher for the 38 projects whose developers acknowledged our notifications.

Project	A	D	P	O	1	2	3	Project	A	D	P	O	1	2	3
GBook MX	60	1	1	Y	Y	Y	Y	SquirrelMail	99	21	1	Y	N	Y	N
AthenaRMS	0	3	1	Y	Y	N	N	PHPMyList	69	1	1	Y	Y	Y	Y
PHPCodeCabine	71	1	2	Y	Y	N	Y	EGroupWare	99	17	1	Y	N	Y	Y
BolinOS	94	7	1	Y	Y	N	N	PHPFriendlyAdmin	87	1	1	Y	Y	N	N
PHP Surveyor	99	6	1	Y	Y	Y	Y	PHP Helpdesk	87	3	1	Y	Y	Y	Y
Booby	90	1	1	N	Y	N	Y	Media Mate	0	2	2	Y	Y	Y	Y
ByteHoard	98	3	1	Y	Y	N	Y	Obelus Helpdesk	22	2	2	N	Y	N	Y
PHPRecipeBook	99	1	1	Y	Y	N	Y	eDreamers	80	5	1	N	Y	N	N
phpLDAPadmin	97	3	1	Y	Y	N	N	Mad.Thought	66	1	3	Y	Y	Y	N
Segue CMS	77	4	1	Y	Y	Y	N	PHPLetter	79	1	1	Y	Y	Y	Y
Moregroupware	99	11	1	Y	Y	Y	N	WebArchive	2	3	1	Y	Y	N	Y
iNuke	0	1	1	Y	Y	Y	N	Nalanda	58	1	1	Y	Y	N	N
InfoCentral	82	9	1	Y	Y	Y	N	Site@School	94	2	1	Y	Y	Y	Y
WebMovieDB	24	3	1	Y	Y	N	N	PHPList	0	1	1	Y	Y	Y	N
TestLink	88	4	1	Y	Y	Y	Y	PHPPgAdmin	98	6	1	Y	Y	N	N
Crafty Syntax	0	1	1	N	Y	N	N	Anonymous Mailer	73	1	1	N	N	Y	Y
ILIAS open source	20	2	4	Y	Y	Y	Y	PHP Support Tickets	0	1	1	N	Y	Y	N
PHP Multiple Newsletters	68	1	1	N	Y	Y	N	Norfolk Household Finan. Manager	0	1	1	N	Y	Y	Y
International Suspect Vigilance Nexus	0	1	1	N	N	Y	N	Tiki CMSGroupware	99	135	2	N	N	Y	Y
								Activity / Total	62	7	1.2	28	33	22	18

A: Project activity
D: Number of Developers
P: Vulnerability Depth
O: Overlooked / unaware
1: Cross-site scripting
2: SQL injection
3: General script injection

Figure 15. The 38 vulnerable projects that have responded to our notifications.

Developer motivation and behavior is outside the scope of this paper, but we did 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 discovered (from the developers' responses) that some of these projects had vulnerabilities that had already been identified and disclosed prior to the present project. For instance, *ByteHoard* had one Bugtaq disclosure report, and *SquirrelMail* had 13 CVE (Common Vulnerability Exposure) records. 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; 515 files were identified as vulnerable. Even with the special features provided by the WebSSARI GUI, manual validation of our results turned out to be a very time-consuming task, because it required investigation into multiple function calls that spanned across multiple files. Fortunately, the PHPXREF [74] that we incorporated into our approach sped up the process. After four days of manual inspection, we concluded that only 361 files were indeed vulnerable—a false positive rate of 29.9 percent. After adding support for type-aware qualifiers, the number of insecure files reported by WebSSARI dropped to 494, 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 sensitive function calls using 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 (57404/1140091) of the 1140091 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 3.6, 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.

Development Status		Current Activity (Activity + Popularity)			
<= Beta	>= Stable	0-25%	26-50%	51-75%	76-100%
31	38	21	6	11	31

Figure 16. Distribution of all 69 vulnerable projects.

Development Status		Current Activity (Activity + Popularity)			
<= Beta	>= Stable	0-25%	26-50%	51-75%	76-100%
10	28	11	0	6	21

Figure 17. Distribution of the 38 vulnerable projects that have responded to our notifications.

7. DISCUSSION

In order to experiment with our proposed algorithm, we have chosen to implement a code walker for PHP. However, by providing other code walker implementations, our approach can be used for other Web programming languages as well. In recognition of the difficulty to develop secure Web application code, popular scripting languages have provided various aids—e.g., Perl's tainted mode and PHP's magic_quotes option. These features offer runtime protection, but are incapable of compile-time bug identification. Perl's tainted mode tracks information flow at runtime, which induces expensive overhead. PHP's magic_quotes option causes the PHP interpreter to automatically escape certain problematic characters within tainted data using backslashes. However, depending on how tainted data is being used, the set of problematic characters differ, and more importantly, so do the means for escaping them. Therefore, although this strategy helps eliminate certain attacks such as SQL injection, it will not work against attacks such as cross-site scripting, where sanitization requires escaping a different set of characters according to HTML character entity references (instead of simply using backslashes).

One weakness of our approach is that it identifies symptoms of errors rather than their causes. Maintaining only the most restrictive type environment at execution path merge points keeps the search space small, yet it also forbids providing counterexample traces. This forces us to insert runtime guards at potentially vulnerable function call sites, with the guards sanitizing the tainted variables before they were used as arguments to call sensitive functions. However, following an initial induction, a single piece of tainted data is capable of triggering a snowballing process of propagation and tainting of other data, with the number of tainted variables growing exponentially as the program executes. A more efficient strategy would be to use an algorithm capable of giving counterexample traces to identify the point where the tainting process begins and to sanitize the data before it propagates. In a following project [39], we addressed this weakness using bounded model checking, which offers counterexamples at a reasonable cost.

8. CONCLUSION

Security remains a major roadblock to universal acceptance of the Web for many kinds of transactions, especially since the recent sharp increase in remotely exploitable vulnerabilities have been attributed to Web application bugs. Scott and Sharp's global protection mechanism [62] [63], AppShield [58], and InterDo [43] offer protection methods that immediately ensure the security of Web applications, but they require careful configuration by experienced administrators [12]. At least four assessment frameworks for Web application security (WAVES [38], AppScan [59], WebInspect [66], and ScanDo [43]) provide black-boxed testing capability for identifying Web application vulnerabilities. Still, testing approaches can never guarantee soundness. Here we have tried to establish an approach that retains the advantages and eliminates the disadvantages of preceding efforts.

Our approach provides immediate protection at a much lower cost than Scott and Sharp's, since validation is restricted to potentially vulnerable sections of code. If WebSSARI detects the use of untrusted data following correct treatment, 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 perform unconditional global validation for every bit of user-submitted data without concerning that the Web application may incorporate the same validation routine, thus resulting in unnecessary overhead. If a Web application utilizes HTTPS for traffic encryption, the associated decrypt-validate-encrypt process may limit scalability. Furthermore, WebSSARI provides protection in the absence of user intervention, as compared with the user expertise required for their 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 [50], Banerjee and Naumann [7], and Pottier and Simonet [56], our approach verifies the most commonly used language for Web application programming, and we incorporate 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; in contrast, we proposed using 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 to flow-sensitive approaches such as Ashcraft and Engler [3] and Shankar et al. [65], we proposed a type-aware lattice model in contrast to their primitive T-U lattice. According to our experimental results, the use of this lattice model helped to reduce false positives by 10.03 percent. Compared to unsound checkers [3] [26] [71] [64] [75] [20] [44] [69], our approach attempts to provide a sound verification framework.

9. ACKNOWLEDGMENT

We deeply appreciate the anonymous reviewers for offering us many valuable comments. We would also like to thank Dr. Bow-Yaw Wang for his useful suggestions. This project was supported in part by the National Science Council, Taiwan under grants

NSC-93-2422-H-001-0001, NSC-92-2219-E-002-019, and NSC-92-2213-E-001-024.

10. REFERENCES

[1] Allen, F. E, Cocke, J. "A Program Data Flow Analysis Procedure." Communications of the ACM, 19(3):137-147, 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 *Proc. 2002 IEEE Symp. Security and Privacy*, pages 131-147, Oakland, California, 2002.

[4] Augustin, L., Bressler, D., Smith, G. "Accelerating Software Development through Collaboration." In *Proc. 24th Int'l Conf. Software Engineering (ICSE2002)*, pages 559-563, Orlando, Florida, May 19-25, 2002.

[5] Ball, T., Rajamani, S. K., "Automatically Validating Temporal Safety Properties of Interfaces." In *Proc. 8th Int'l SPIN Workshop on Model Checking of Software (SPIN'01)*, pages 103-122, volume LNCS 2057, Toronto, Canada, May 19-21, 2001. Springer-Verlag.

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

[7] Banerjee, A., Naumann, D.A. "Secure Information Flow and Pointer Confinement in a Java-Like Language." In: *Proc. 15th Computer Security Foundations Workshop (CSFW2002)*, pages 239-253, Nova Scotia, Canada, 2002.

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

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

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

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

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

[13] Chen, H., Wagner, D., "MOPS: an Infrastructure for Examining Security Properties of Software." In *Proc. 9th ACM Conf. Computer and Communications Security (CCS2002)*, pages 235-244, Washington, DC, Nov 18-22, 2002.

[14] Cousot, P., Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Constructions or Approximation of Fixpoints." In *Conference Record of the Fourth ACM Symp. Principles of Programming Languages (POPL'77)*, pages 238-252, 1977.

[15] 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 *Proc. 7th USENIX Security Conference (USENIX'98)*, pages 63-78, San Antonio, Texas, Jan 1998.

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

[17] 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.

[18] Darvas, A., Hähnle, R., Sands, D. "A Theorem Proving Approach to Analysis of Secure Information Flow." In *Proc. Workshop on Issues in the Theory of Security (WITS'03)*, Warsaw, Poland, Apr 5-6, 2003.

[19] Das, M., Lerner, S., Seigle, M. "ESP: Path-Sensitive Program Verification in Polynomial Time." In *Proc. 2002 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI2002)*, pages 57-68, Berlin, Germany, 2002.

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

[21] DeLine, R. Fahndrich, M. "Enforcing High-Level Protocols in Low-Level Software." In *Proc. ACM SIGPLAN 2001 Conf. Programming Language Design and Implementation (PLDI2001)*, pages 59-69, Snowbird, Utah, 2001.

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

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

[24] Dharmapurikar, S., Krishnamurthy, P., Sproull, T., and Lockwood, J. "Deep Packet Inspection Using Parallel Bloom Filters." In *Proc. 11th Symp. High Performance Interconnects (HOTI'03)*, pages 44-51, Stanford, California, 2003.

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

[26] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. "Extended Static Checking for Java." In *Proc. 2002 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI2002)*, pages 234-245, volume 37(5) of ACM SIGPLAN Notices, Berlin, Germany, Jun 2002.

[27] Foster, J. S., Fähndrich, M., Aiken, A. "A Theory of Type Qualifiers." In *Proc. ACM SIGPLAN 1999 Conf. Programming Language Design and Implementation (PLDI'99)*, pages 192-203, volume 34(5) of ACM SIGPLAN Notices, Atlanta, Georgia, May 1-4, 1999.

[28] Foster, J., Terauchi, T., Aiken, A. "Flow-Sensitive Type Qualifiers." In *Proc. ACM SIGPLAN 2002 Conf. Programming Language Design and Implementation (PLDI2002)*, pages 1-12, Berlin, Jun 2002.

[29] Gagnon, E. M., Hendren, L. J. "SableCC, an Object-Oriented Compiler Framework." In *Proc. 1998 Conf. Technology of Object-Oriented Languages and Systems (TOOLS-98)*, pages 140-154, Santa Barbara, California, Aug 3-7, 1998.

[30] Goguen, J. A., Meseguer, J. "Security Policies and Security Models." In *Proc. IEEE Symp. Security and Privacy*, pages 11-20, Oakland, California, Apr 1982.

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

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

[33] Hallem, S., Chelf, B., Xie, Y., Engler, D. "A System and Language for Building System-Specific, Static Analyses." In *Proc. ACM SIGPLAN 2002 Conf. Programming Language Design and Implementation*, pages 69-82, Berlin, Germany, 2002.

[34] Hecht, M. S., Ullman, J. D. "Analysis of a Simple Algorithm For Global Flow Problems." In *Conference Record of the First ACM Symp. Principles of Programming Languages (POPL'73)*, pages 207-217, Boston, Massachussets, 1973.

[35] Henglein, F. "Dynamic Typing." In *Proc. Fourth European Symp. Programming (ESOP'92)*, pages 233-253, volume LNCS 582, Rennes, France, Feb 1992. Springer-Verlag.

[36] 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.

[37] Holzmann, G. J. “The Logic of Bugs.” In *Proc. 10th ACM SIGSOFT Symp. Foundations of Software Engineering (FSE-10)*, pages 81-87, Charleston, South Carolina, 2002.

[38] Huang, Y. W., Huang, S. K., Lin, T. P., Tsai, C. H. “Web Application Security Assessment by Fault Injection and Behavior Monitoring.” In *Proc. Twelfth Int’l World Wide Web Conference (WWW2003)*, 148-159, Budapest, Hungary, May 21-25, 2003.

[39] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., Kuo, S. Y. “Verifying Web Applications Using Bounded Model Checking.” In: *Proc. 2004 Int’l Conf. Dependable Systems and Networks (DSN2004)*, Florence, Italy, Jun 28-Jul 1, 2004.

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

[41] Jensen, T., Le Metayer, D., Thorn, T. “Verification of Control Flow Based Security Properties.” In *Proc. 20th IEEE Symp. Security and Privacy*, pages 89-103, IEEE Computer Society, New York, USA, 1999.

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

[43] Kavado, Inc. “InterDo Version 3.0.” Kavado Whitepaper, 2003.

[44] Larochelle, D., Evans, D. “Statically Detecting Likely Buffer Overflow Vulnerabilities.” In *Proc. 10th USENIX Security Symposium (USENIX’01)*, Washington, D.C., Aug 2001.

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

[46] Meier, J.D., Mackman, A., Vasireddy, S. Dunner, M., Escamilla, R., Murukan, A. “Improving Web Application Security—Threats and Countermeasures.” Microsoft Corporation, 2003.

[47] 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

[48] 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.

[49] 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.

[50] Myers, A. C. “JFlow: Practical Mostly-Static Information Flow Control.” In *Proc. 26th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL’99)*, pages 228-241, San Antonio, Texas, 1999.

[51] Necula, G. C. “Proof-Carrying Code.” In *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL’97)*, pages 106-119, Paris, France, Jan 1997.

[52] Necula, G. C., McPeak, S., Weimer, W. “CCured: Type-Safe Retrofitting of Legacy Code.” In *Proc. 29th Annual ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL2002)*, pages 128-139, Portland, Oregon, 2002.

[53] Orbaek, P. “Can You Trust Your Data?” In *Proc. 1995 TAPSOFT/FASE Conference*, pages 575-590, volume LNCS 915, Aarhus, Denmark, May 1995. Springer-Verlag.

[54] OWASP. “The Ten Most Critical Web Application Security Vulnerabilities.” OWASP Whitepaper, version 1.0, 2003.

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

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

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

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

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

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

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

[62] Scott, D., Sharp, R. “Abstracting Application-Level Web Security.” In: *Proc. 11th Int’l Conf. World Wide Web (WWW2002)*, pages 396-407, Honolulu, Hawaii, May 17-22, 2002.

[63] Scott, D., Sharp, R. “Developing Secure Web Applications.” IEEE Internet Computing, 6(6), 38-45, Nov 2002.

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

[65] Shankar, U., Talwar, K., Foster, J. S., Wagner, D. “Detecting Format String Vulnerabilities with Type Qualifiers.” In *Proc. 10th USENIX Security Symposium (USENIX’02)*, pages 201-220, Washington DC, Aug 2002.

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

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

[68] 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.

[69] 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 (ACSAC’00)*, New Orleans, Louisiana, Dec 11-15, 2000.

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

[71] Wagner, D., Foster, J. S., Brewer, E. A., Aiken, A. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.” In *Proc. 7th Network and Distributed System Security Symposium (NDSS2000)*, pages 3-17, San Diego, California, Feb 2000.

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

[73] Walker, D. “A Type System for Expressive Security Policies.” In *Proc. 27th Symp. Principles of Programming Languages (POPL’00)*, pages 254-267, ACM Press, Boston, Massachusetts, Jan 2000.

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

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

[76] Witten, B., Landwehi, C., Caloyannides, M., “Does Open Source Improve System Security?” IEEE Software, 18(5):57-61, 2001.

[77] 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