Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Networks 48 (2005) 739–761

Computer Networks

www.elsevier.com/locate/comnet

# A testing framework for Web application security assessment ☆

Yao-Wen Huang [a,b,*], Chung-Hung Tsai [b], Tsung-Po Lin [b], Shih-Kun Huang [a,c], D.T. Lee [a,b,c], Sy-Yen Kuo [a,b]

[a] *Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan*
[b] *Institute of Information Science, Academia Sinica, Taipei 115, Taiwan*
[c] *Department of Computer Science and Information Engineering, National Chiao-Tung University, Hsinchu 300, Taiwan*

Available online 12 February 2005

**Abstract**

The rapid development phases and extremely short turnaround time of Web applications make it difficult to eliminate their vulnerabilities. Here we study how software testing techniques such as fault injection and runtime monitoring can be applied to Web applications. We implemented our proposed mechanisms in the Web Application Vulnerability and Error Scanner (WAVES)—a black-box testing framework for automated Web application security assessment. Real-world situations are used to test WAVES and to compare it with other tools. Our results show that WAVES is a feasible platform for assessing Web application security.
© 2005 Elsevier B.V. All rights reserved.

*Keywords:* Web application testing; Security assessment; Fault injection; Black-box testing; Complete crawling

## 1. Introduction

On Feb 2, 2000, CERT issued an advisory [16] 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 [40]. 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

☆ Supported by the NCERT project (No. 910444) of Institute for Information Industry, sponsored by Ministry of Economic Affairs, and by the National Science Council under the grants No. NSC 92-2422-H-001-004, NSC-93-2213-E-002-122, NSC-93-2213-E-001-013, NSC-93-2422-H-001-0001, and NSC-93-2752-E-002-005-PAE.

* Corresponding author. Address: Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan.
*E-mail addresses:* ywhuang@openwaves.net (Y.-W. Huang), chtsai@openwaves.net (C.-H. Tsai), lancelot@iis.sinica.edu.tw (T.-P. Lin), skhuang@iis.sinica.edu.tw (S.-K. Huang), dtlee@iis.sinica.edu.tw (D.T. Lee), sykuo@cc.ee.ntu.edu.tw (S.-Y. Kuo).

1389-1286/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2005.01.003

by Ohmaki [43] found that almost 80% of all e-commerce sites in Japan were still vulnerable to XSS. A search on Google News 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 [70] and Yahoo! Mail [32], both of which are popular web-based email services.

Scott and Sharp [58,59] have asserted that Web application vulnerabilities are (a) inherent in Web application codes; 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 [18]. However, when network and host-level entry points are relatively secure, the public interfaces of Web applications become the focus of attacks [18].

Recently, efforts on automated security verification of C programs have yielded promising results. In our project *WebSSARI v.1* (Web application Security Analysis and Runtime Inspection) [24], we adopted a typestate-based algorithm for identifying vulnerabilities in PHP code. In *WebSSARI v.2* [25], we showed how this strategy can be improved using bounded model checking. Though our results show that such white-box verification can be successfully used for automated Web application security assessment, they have several drawbacks. One major drawback is their need for source codewhich in many cases may not be easily available. Another is that they verify against simulated runtime behaviors based on program abstraction, while whether or not the abstraction correctly reflects the actual program is left in question. Therefore, while these techniques fail to adequately consider the runtime behavior of Web applications, it is generally agreed that the massive number of runtime interactions between various components is what makes Web application security such a challenging task [28,58].

In this paper, we describe our black-box testing framework for Web application security assessment. The main difficulty in designing a Web application testing framework lies in providing efficient interface mechanisms. Since Web applications interact with users behind browsers and act according to user input, such interfaces must have the ability to mimic both the browser and the user. In other words, the interface must process content that is meant to be rendered by browsers and later interpreted by humans. Our interface takes the form of a crawler, which allows for a black-box, dynamic analysis of Web applications. Using a *complete crawling* mechanism, a reverse engineering of a Web application is performed to identify all data entry points. Then, with the help of a self-learning injection knowledge base, fault injection techniques are applied to detect SQL injection vulnerabilities. Using our proposed *Topic Model*, the knowledge base selects the best injection patterns according to experiences learned through previous injection feedback, and then expands the knowledge base as more pages are crawled. Both previous experiences and knowledge expansion contribute to the generation of better injection patterns. We also propose a novel reply analysis algorithm in order to help the crawler interpret injection replies. By improving the observability [72] of the Web application being tested, the algorithm helps facilitate a *deep injection* mechanism that eliminates false negatives.

To imitate real-world interactions with Web applications, our crawler is equipped with the same capabilities as a full-fledged browser, thus making it vulnerable to malicious scripts that may have been inserted into a Web application via cross-site scripting. Since a malicious script that is capable of attacking an interacting browser is also capable of attacking the crawler, a secure execution environment (SEE) that enforces an anomaly detection model was built around the crawler. During the reverse engineering phase, all pages of a Web application are loaded into the crawler and executed. Input stimuli (i.e., simulated user events) are generated by the crawler to test the behavior of the page's dynamic components. Any abnormal behavior will cause the SEE to immediately halt the crawler and audit the information. Thus, while offering self-protection, this layer also detects malicious scripts hidden inside Web applications.

## 2. Web application vulnerabilities

In their attempt to address Web application security, Scott and Sharp [58,59] selected three types of vulnerabilities they believed to be particularly importantform modification, SQL injection, and XSS. They also suggested that form modification is often used in conjunction with other forms of attacks, for example, SQL injection. The other two types of vulnerabilities, SQL injection (which resembles the C format string vulnerability [63]) and XSS, indeed account for the majority of Web application vulnerabilities [18]. Here we chose SQL injection and cross-site scripting vulnerabilities as our primary detection targets. Our algorithm also detects SQL injection and XSS vulnerabilities that is exploited via form modification, but we consider form modification as a particular means to exploit the other two types of vulnerabilities rather than a separate type of vulnerability. We will briefly describe the two vulnerabilities, followed by our proposed detection mechanisms.

### *2.1. SQL injection*

Web applications often use data read from a client to construct database queries. If the data is not properly processed prior to SQL query construction, malicious patterns that result in the execution of arbitrary SQL or even system commands can be injected [5].

Consider the following scenario: a Web site includes a form with two edit boxes in its login.html to ask for a username and password. The form declares that the values of the two input fields should be submitted with the variables strUserName and strPassword to login.cgi, which includes the following code:

```
SQLQuery = "SELECT*FROM Users WHERE
(UserName = '" + strUserName + "') AND
(Password = '" + strPassword + "');"
If GetQueryResult(SQLQuery) = 0 Then bAuthenticated = false; Else bAuthenticated = true;
```

If a user submits the username "Wayne" and the password "0308Wayne," the SQLQuery variable is interpreted as:

```
"SELECT* FROM Users WHERE (strUserName = 'Wayne') AND (Password =
'0308Wayne');"
```

GetQueryResult() is used to execute SQLQuery and retrieve the number of matched records. Note that user inputs (stored in the strUserName and strPassword variables) are used directly in SQL command construction without preprocessing, thus making the code vulnerable to SQL injection attacks. If a malicious user enters the following string for both the UserName and Password fields:

```
X' OR 'A' = 'A
```

then the SQLQuery variable will be interpreted as:

```
"SELECT* FROM Users WHERE (strUserName = 'X' OR 'A' = 'A') AND (Password = 'X'
OR 'A' = 'A');"
```

Since the expression 'A' = 'A' will always be evaluated as TRUE, the WHERE clause will have no actual effect, and the SQL command will always be the equivalent of "SELECT* FROM Users". Therefore, GetQueryResult() will always succeed, thus allowing the Web application's authentication mechanism to be bypassed.

### *2.2. SQL injection detection*

Our approach to SQL injection detection entails fault injection-a dynamic analysis process used for software verification and software security assessment. For the latter task, specially crafted malicious input patterns are deliberately used as input data, allowing developers to observe the behavior of the software under attack. Our detection model works in a similar manner—that is, we identify vulnerabilities in Web applications by observing the output resulting from the specially prepared SQL injection patterns.

Similar to other research on Web site testing and analysis [8,51,55], we adopted a black-box approach in order to analyze Web applications externally without the aid of source code. 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. Such a remote, black-box security testing tool for Web applications is also called a *Web application security scanner (WSS)*. Since more available information on a program (e.g., function specifications or source code) results in the possibility to develop more thorough software testing processes for that program, it is necessary to define what a WSS is. Here we will define a WSS as a testing platform posed as an outsider (i.e., as a public user) to the target application. Therefore, WSSs 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, environment variable modification, and local files access) is not possible.
3. The testing process must be automated and testing a new target system should not require extensive human participation in test case generation.

In white-box testing, source code analysis provides critical information for effective test case generation [50]. In black-box testing (in other words, when source code is unavailable), an information-gathering approach is to reverse-engineer executable code. To perform this task, we designed a crawler to crawl the Web application—an approach adopted in many Web site analysis [8,51,55] and reverse engineering [52–54] studies. We designed our crawler to discover all pages in a Web site that contain HTML forms, since forms are the primary data entry points in most Web applications. From our initial tests, we learned that ordinary crawling mechanisms normally used for indexing purposes [13,17,36,38,61,67] are unsatisfactory in terms of thoroughness. Many pages within Web applications currently contain such dynamic content as Javascripts and DHTML. 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 show that all traditional crawlers (which use static parsing and lack script interpretation abilities) tend to skip pages in Web sites that have these features (see Section 3). In both security assessment and fault injection, completeness is an important issue–that is, all data entry points must be correctly identified. No attempt was made to exhaust input space, but we did emphasize the importance of comprehensively identifying all data entry points, since a single unidentified link would nullify the tests conducted for all other links.

### *2.2.1. Complete crawling*

In order to improve coverage, we looked at ways that HTML pages reveal the existence of other pages or entry points, 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/topframe.htm"⟩
3. Meta refresh redirections.
   Ex: ⟨meta http-equiv = "refresh" 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 [19].
8. Form submissions.

We established a sample site to test several commercial and academic crawlers, including Teleport

[67], WebSphinx [38], Harvest [13], Larbin [61], Web-Glimpse [36], and Google. None were able to crawl beyond the fourth level of revelation—about one-half of the capability of the WAVES crawler. In order to detect revelations 5 and 6, WAVES' crawler incorporates a full-fledged browser (detailed in Section 3), allowing it to interpret Javascripts. Revelation 7 also refers to link-revealing Javascripts, but only following an onClick, onMouseOver, or similar user-generated event. As described in Section 2.4, WAVES performs an event-generation process to stimulate the behavior of active content. This not only allows WAVES to detect malicious components but, together with WAVES' Javascript interpretation ability, allows WAVES to detect revelation 7. 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 crawlers capable of handling level-7 revelations. Also note that even though WAVES' test set generation algorithm's (detailed in Section 2.2.3) main purpose is to produce quality test sets, the same knowledge can also be used during the crawl process. When a crawler encounters a form, it uses the algorithm for automated form submissions (revelation 8).

During the reverse engineering process, HTML pages are parsed with a Document Object Model (DOM) [4] parser (provided by the incorporated browser component), and HTML forms are parsed and stored in XML format. These forms contain such data as submission URLs, available input fields and corresponding variables, input limits, default values, and available options.

Upon completion of the reverse engineering process, an attempt was made to inject malicious SQL patterns into the server-side program that processes the form's input. We referenced the existing literature on SQL injection techniques to create a set of SQL injection patterns [5]. The first step is to determine the variable on which to place these patterns. A typical example of HTML form source code is presented in Fig. 1:

An example of a URL generated by submitting the form in Fig. 1 is:

http://waves.net/~lancelot/submit.cgi?strUserName = Wayne&strPassword = 0308Wayne

Note that two variables were submitted. If the SQL injection pattern is placed on strUserName, then the submitted URL will appear as:

http://waves.net/~lancelot/submit.cgi?strUserName = X'OR'A' = 'A&strPassword=

Depending on the SQL injection technique being used, patterns are placed on either the first or last variable. If the server-side program does not incorporate a pre-processing function to filter malicious input, and if it does not validate the correctness of input variables before constructing and executing the SQL command, the injection will be successful.

However, if the server-side program detects and filters malicious patterns, or if the filtering mechanism is provided on a global scale (e.g., [58]), then injection will fail, and a false negative report will be generated. Many server-side programs execute validation procedures prior to performing database access. An example of a typical validation procedure added to the code presented in Fig. 1 is shown below:

```
If Length(strUserName < 3) OR Length(strUserName > 20) Then
OutputError(''Invalid User Name'') Else
If Length(strPassword < 6) OR Length(strPassword) > 11) Then
```

```
<Form action="submit.cgi" method="GET">
strUserName: <Input type="text" name="strUserName" size="20" maxlength="20"><br>
strPassword: <Input type="password" name="strPassword" size="20" maxlength="20"><br>
··· (skipped)
```

Fig. 1. A typical example of HTML form source code.

```
OutputError(''Invalid Password'') Else Begin
SQLQuery = ''SELECT* FROM Users WHERE
UserName = ''' + strUserName + ''AND Password
= ''' + strPassword + ''';''
If GetQueryResult(SQLQuery) = 0 Then
bAuthenticated = false;
Else bAuthenticated = true;
End;
```

For the above code, our injection URL will fail because it lacks a password. The code requires that the variable strPassword carry text containing between 6 and 11 characters; if a random 6-11 character text is assigned to strPassword, injection will still succeed. We propose the use of a "deep injection" mechanism to eliminate these types of false negatives.

### 2.2.2. The topic model

To bypass the validation procedure, the Injection Knowledge Manager (IKM) must decide not only on which variable to place the injection pattern, but also how to fill other variables with potentially valid data. Here we looked at related research in the area of automated form completion—that is, the automatic filling-out of HTML forms. A body of information approximately 500 times larger than the current indexed Internet is believed to be hidden behind query interfaces [9] for example, patent information contained in the United States Patent and Trademark Office's Web site [69]. Since only query (and not browsing) interfaces are provided, these types of document repositories cannot be indexed by current crawling technologies. To accomplish this task, a crawler must be able to perform automatic form completion and to send queries to Web applications. These crawlers are referred to as "deep crawlers" [9] or "hidden crawlers" [27,33,48]. Here we adopted an approach similar to [48], but with a topic model that enhances submission correctness and provides a self-learning knowledge expansion model.

Automated form completion requires the selection of syntactically or even semantically correct data for each required variable. For example, the variable names "strUserName" and "strPassword" shown in Fig. 1 reveal both syntactic and semantic information. The "str" prefix indicates text as the required data type, and the "UserName" suggests the text semantics (i.e., a person's name). Supplying the IKM with knowledge to provide a valid input for each variable requires the design and implementation of a self-learning knowledge base.

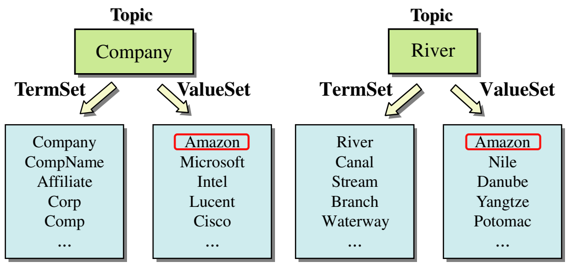


Fig. 2. The Topic model.

We designed our knowledge base using our proposed *Topic Model* (see Fig. 2). At the model's center is the "topic"e.g., "Human Names," "Company Names," "River Names," "Star Names," and "Addresses." In order to describe the model, we define the following notations.

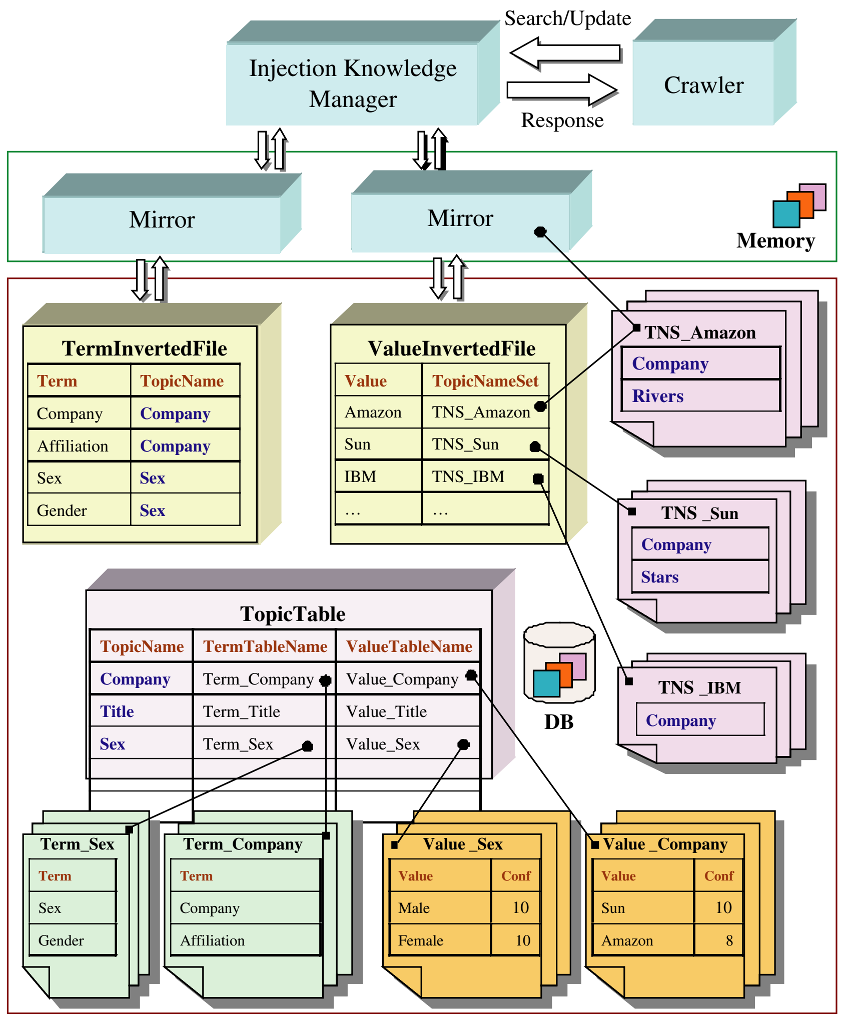
1. $\hat{t} \rightarrow \frown{p}$ if term $\hat{t}$ indicates topic $\overset{\frown}{p}$; for example, terms "sex" and "gender" both indicates topic "sex," and terms "company" and "affiliation" both indicates topic "company."
2. $\tilde{v} \mapsto \overset{\frown}{p}$ if value $\tilde{v}$ is valid for topic $\overset{\frown}{p}$; for example, values "Arnold" and "Wayne" are both valid for topic "first name."
3. $conf(\tilde{v}, \overset{\frown}{p})$ denotes the *confidence value (CV)* of $\tilde{v}$ in respect to $\overset{\frown}{p}$; that is, the likelihood of $\tilde{v}$ being a valid value for $\overset{\frown}{p}$. Each value is assigned an initial CV of 1, and CVs of values involved in an injection iteration are adjusted according to the injection's output (using the Feedback algorithm described below).

We define a topic $\overset{\frown}{p} = \{T(\overset{\frown}{p}), V(\overset{\frown}{p})\}$, where $Termset(\overset{\frown}{p}) = \{\hat{t} | \hat{t} \rightarrow \overset{\frown}{p}\}$, and $Valueset(\overset{\frown}{p}) = \{(\tilde{v}, \mathrm{conf}(\tilde{v}, \overset{\frown}{p})) | \tilde{v} \mapsto \overset{\frown}{p}\}$. Our knowledge base $KB = \{\overset{\frown}{p}_1, \overset{\frown}{p}_2, \ldots, \overset{\frown}{p}_n\}$ is therefore a collection of topics. We note that for any pair of topics $\overset{\frown}{p}_1$ and $\overset{\frown}{p}_b$, $|Valueset(\overset{\frown}{p}_a) \cap \mathrm{Valueset}(\overset{\frown}{p}_b)| \geqslant 0$ (in other words, values can be associated with more than one topic). Thus, the value "Amazon" may appear in both $Valueset(\widehat{company})$ and $Valueset(\widehat{river})$, and "Sum" may appear in both $Valueset(\widehat{company})$ and $Valueset(\widehat{star})$.

### 2.2.3. Test set generation and output analysis algorithms

In our *KB* implementation (see Fig. 3), we use two inverted files to improve search speed. We construct $TermInvertedFile = \{(\hat{t}, \overset{\frown}{p}) | \overset{\frown}{p} \in KB$ and $\hat{t} \rightarrow \overset{\frown}{p}\}$. We index all terms within the inverted file, so that $TermInvertedFile.Topic(\hat{t})$ promptly returns the topic associated with term $\hat{t}$. We also construct $ValueInvertedFile = \{(\tilde{v}, P) | \forall \overset{\frown}{p} \in KB$ and $\hat{v} \mapsto \overset{\frown}{p}, \overset{\frown}{p} \in P\}$. Similarly, we index all values within the inverted file so that $TermInvertedFile.Topic(\tilde{v})$ promptly returns the topic associated with $\tilde{v}$.

Using our *KB*, The IKM implements four algorithms Get_Topic(), Get_Value(), Expand_Values() and Feedback(). When confronted with a text box that carries no default value, the crawler calls IKM's Get_Value($\hat{t}$) to retrieve the best possible guess, where $\hat{t}$ is the term (variable name or descriptive keyword) associated with the text box. Internally, Get_Value() utilizes Get_Topic(), which checks whether a topic can be associated with $\hat{t}$. Get_Topic() and Get_Value() are described as follows:



TermInvertedFile

| Term | TopicName |
|---|---|
| Company | Company |
| Affiliation | Company |
| Sex | Sex |
| Gender | Sex |

ValueInvertedFile

| Value | TopicNameSet |
|---|---|
| Amazon | TNS_Amazon |
| Sun | TNS_Sun |
| IBM | TNS_IBM |
| … | … |

TNS_Amazon

| |
|---|
| Company |
| Rivers |

TNS _Sun

| |
|---|
| Company |
| Stars |

TNS _IBM

| |
|---|
| Company |

TopicTable

| TopicName | TermTableName | ValueTableName |
|---|---|---|
| Company | Term_Company | Value_Company |
| Title | Term_Title | Value_Title |
| Sex | Term_Sex | Value_Sex |

Term_Sex

| Term |
|---|
| Sex |
| Gender |

Term_Company

| Term |
|---|
| Company |
| Affiliation |

Value _Sex

| Value | Conf |
|---|---|
| Male | 10 |
| Female | 10 |

Value _Company

| Value | Conf |
|---|---|
| Sun | 10 |
| Amazon | 8 |

Fig. 3. The self-learning knowledge base model.

```
Get_Topic()
Input: $\hat{t}$—The newly encountered variable name or descriptive keyword.
Output: $\widehat{p}$—Name of the topic containing Input-Term, if any.

1. Find $\hat{s}$ that is a term most similiar (having the nearest edit distance) to $\hat{t}$:
   a. $\forall \hat{t}_i \in TermInvertFile$, $Similarity(\hat{t}_i) = 1/NearestEditDist(\hat{t}_i, \hat{t})$.
   b. $max = Max(Similarity(\hat{t}_1), \ldots, Similarity(\hat{t}_n))$;
2. If no similar ones found, return "not found," else return the topic containing the similar term:
   if $max < \rho$ then
      $\widehat{p}$ ="notfound"; return;
   else
      $\widehat{p} = TermInvertFile.Topic(\hat{t}_{max})|$
      $Similarity(\hat{t}_{max})$ equals $max$; return $\widehat{p}$
```

```
Get_Value()
Input: $\hat{t}$—The newly encountered variable name or descriptive keyword.
Output: $\tilde{v}$—The candidate value having the highest confidence.

1. Check whether $\hat{t}$ can be associated with a topic:
   a. Get_Topic($\hat{t}, \widehat{p}_{match}$)
   b. if $\widehat{p}_{match}$ equals "not found" then $\tilde{v}$="not found"; return;
2. Retrieve the candidate having the highest confidence:
   a. $\forall V_i \in ValueSet(\widehat{p}_{match})$,
      $max = Max(Conf(V_1), \ldots, Conf(V_n))$;
   b. $\tilde{v} = \tilde{v}_i | \tilde{v}_i \in ValueSet(\widehat{p}_{match})$,
      $Conf(\tilde{v}_i)$ equals $max$
```

Get_Topic() uses a simple string similarity-matching algorithm to compute $\hat{t}$'s nearest edit distances to every term from every topic contained in the knowledge base. This approach ensures that similar phrases (e.g., "User_Name" and "User-Name") are marked as having a short distance. To reduce computation complexity, matching is performed using the *TermInvertedFile* stored in memory (Fig. 3). A minimum threshold $\rho$ is set so that Get_Topic() may fail (indicating that no relevant topic is found).

After an associated topic $\widehat{p}_{match}$ the candidate with the highest confidence (denoted $\tilde{v}$). If two or more candidates with the same confidence are identified, one is randomly selected. $\tilde{v}$ is then returned to the crawler, which calls Get_Value() iteratively until it has enough values to construct a deep SQL injection URL. Following an injection, the crawler calls Feedback() to supply the IKM with feedback on the successfulness of the injection. Confidence is adjusted for each value involved in the injection session.

The key terms used in the process just described consist of variable names gathered from the HTML form's source code. Though programmers with good practices are likely to follow proper naming conventions, doing so is not considered as mandatory, and poor-looking codes will not affect a form's appearance or functionality. For this reason, it is not possible to rely solely on these variable names to provide descriptive (syntactic or semantic) information regarding input fields. Raghavan [48] has proposed an algorithm called LITE (Layout-based Information Extraction Technique) to help identify input field semantics. In LITE, the HTML is sent to an approximate DOM parser, which calculates the location of each DOM element rendered on the screen; text contained in the element nearest the input field is considered descriptive. We took a similar approach: our crawler is equipped with a fully functional DOM parser, and thus contains knowledge on the precise layout of every DOM component. While variable names are extracted, the crawler also calculates the square-distance between input fields and all other DOM components. The text from the nearest component is extracted as a second descriptive text. Keywords are further extracted from the text by filtering stop words. The keywords are used to call Get_Value() if the first call using the variable name fails.

After the query and ranking mechanisms are in place and the IKM begins to feed high-confidence terms to the crawlers, the next issue involves populating and automatically expanding the knowledge base. The IKM primarily relies on option lists found in HTML forms for the expansion of $S_{\text{Value}}$. Such option lists are rendered as "Combo

Boxes" on the screen; when clicked, they present drop-down menus containing available options (e.g., a list of countries to be included in a registration form). When requesting data that has a fixed set of possible values such as the country name in an address field, an option list is a commonly chosen input method.

When the crawler confronts an input variable with an attached option list, it calls Expand_Values(InputTerm, PredefinedValues), where InputTerm is the variable name and PredefinedValues the associated options list. According to InputTerm and PredefinedValues, Expand_Values() expands the knowledge base. We define Expand_Values() as follows:

```
Expand_Values ()
Inputs: $\hat{t}$—The newly encountered term (variable name or descriptive keyword).
$V_{predefined}$—A value set resembling the options list associated with $\hat{t}$.

1. Check whether $\tilde{t}$ can be associated with a topic:
   a. Get_Topic $(\hat{t}, \widehat{p}_{match})$;
   b. if $\widehat{p}_{match}$ not equal to "not found" then goto step (3)
2. $\hat{t}$ not found in knowledge base, try to add $\hat{t}$
   a. try to find some value set $V_{similiar}$ that resembles $V_{predefined}$:
      $\forall \widehat{p_i} \in KB, V_i = \{\hat{v}_j | \hat{v}_j \in V_{predefined}$ and $\exists \hat{v}_k \in$ $ValueIntertedFile$ such that $Similarity(\hat{v}_j, \hat{v}_k) > \rho$ and $ValueInvertedFile.Topic(\hat{v}_k)$ equals $\hat{p}_j\}$
      $Score(\hat{p}_i) = |V_i| / |ValueSet(\hat{p}_i)|$
   b. $max = Max(Score(\hat{p}_0), Score(\hat{p}_1), \ldots, Score(\hat{p}_n))$
   c. If $max$ equals to 0 then return
   d. $\exists \hat{p}_{match}$ such that $Score(\hat{p}_{match})$ equal $max$
      $TermSet(\hat{p}_{match}) = TermSet(\hat{p}_{match}) \cup \{\hat{t}\}$
      $TermSet\ (\hat{p}_{match})$ is thus expanded.
3. $\hat{t}$ associated with (if step (2) skipped) or added (if step (2) taken) to a topic. For the topic $\hat{p}_{match}$ containing $\hat{t}$, expand $ValueSet(\hat{p}_{match})$:
   a. $ValueSet(\hat{p}_{match}) = ValueSet(\hat{p}_{match}) \cup V_{predefined}$
      $ValueSet(\hat{p}_{match})$ is thus expanded.
```

If Expand_Values() is able to associate a topic with InputTerm, it appends to the topic's value set all possible values extracted from the newly encountered option list. This enabled the expansion of the value sets as pages are crawled. To expand the term sets, Expand_Values() searches the *ValueInvertedFile* and identifies an existing value set that is most similar to the input set PredefinedValues. If one is identified, InputTerm is added to the term set of the topic of the matched $S_{\text{Value}}$. In the following example, assume that for the topic *Company*, *TermSet*(*Company*) = {"Company", "Firm"} (note that "Affiliation" is not included) and *ValueSet*(*Company*) = {"IBM," "HP," "Sun," "Lucent," "Cisco"}. Then assume that a crawler encounters an input variable "Affiliation" that is associated with an option list $V_{predefined}$ = {"HP," "Lucent," "Cisco," "Dell"}. The crawler calls Expand_Values() with "Affiliation" and $V_{predefined}$. After failing to find a nearest term for "Affiliation," the IKM notes that *ValueSet*(*Company*) is very close to $V_{predefined}$, and inserts the term "Affiliation" into *TermSet*(*Company*) and the value $ValueSet(Company) - V_{predefined}$ = {"Dell"} into *ValueSet*(*Company*). In this scenario, both *TermSet*(*Company*) and *ValueSet*(*Company*) are expanded (all three steps in Expand_Values() are taken).

Here we will describe the mechanism for observing injection results. Injections take the form of HTTP requests that trigger responses from a Web application. Fault injection observability is defined as the probability that a failure will be noticeable in the output space [72]. The observability of a Web application's response is extremely low for autonomous programs, which presents a significant challenge when building hidden crawlers [48]. After submitting a form, a crawler receives a reply to be interpreted by humans; it is difficult for a crawler to interpret whether a particular submission has succeeded or failed. Raghavan [48,49] addresses the problem with a variation of the LITE algorithm: the crawler examines the top-center part of a screen for predefined keywords that indicate errors (e.g., "invalid," "incorrect," "missing," and "wrong"). If one is found, the previous request is considered as having failed.

For successful injections, observability is considered high because the injection pattern causes a database to output certain error messages (see

Section 2.2.4 for a list of injection patterns and error messages). By scanning for key phrases in the replied HTML (e.g. "ODBC Error"), a crawler can easily determine whether an injection has succeeded. However, if no such phrases are detected, the crawler is incapable of determining whether the failure is caused by an invalid input variable, or if the Web application filtered the injection and therefore should be considered invulnerable. To resolve this problem, we propose a simple yet effective algorithm called negative response extraction (NRE). If an initial injection fails, the returned page is saved as $R_1$. The crawler then sends an intentionally invalid request to the targeted Web application for instance, a random 50-character string for the UserName variable. The returned page is retrieved and saved as $R_2$. Finally, the crawler sends to the Web application a request generated by the IKM with a high likelihood of validity, but without injection strings. The returned page is saved as $R_3$. $R_2$ and $R_3$ are then compared using WinMerge [73], an open-source text similarity tool.

The return of similar $R_2$ and $R_3$ pages raises one of two possibilities: (a) no validation algorithm was enforced by the Web application, therefore both requests succeeded; or (b) validation was enforced and both requests failed. In the first situation, the failure of $R_1$ allows for the assumption that the Web application is not vulnerable to the injection pattern, even though it did not validate the input data. In the second situation, the crawler enters an $R_3$ regeneration and submission loop. If a request produces an $R_3$ that is not similar to $R_2$, it is assumed to have bypassed the validation process; in such cases, a new SQL injection request is generated based on the parameter values used in the new, successful $R_3$. If the crawler still receives the same reply after ten loops, it is assumed that either (a) no validation is enforced but the application is invulnerable, or (b) a tight validation procedure is being enforced and automated completion has failed. Further assuming under this condition that the Web application is invulnerable induces a false negative (discussed in Section 5 as $P(F_L|V,D)$). If an injection succeeds, it serves as an example of the IKM learning from experience and eventually producing a valid set of values. Together with the self-learning knowledge base, NRE makes a deep injection possible. A list of all possible reply combinations and their interpretations are presented in Fig. 4.

Combination 2 indicates an NRE failure, since WAVES is unable to interpret the injection output. Combinations 1, 3, 4 and 5 all suggest a deterministic test result. Since only combinations 3 and 5 involve an $R_3$ whose values are considered valid by the target entry point, Feedback() is called only for these two combinations.

The crawler's SQL_Injection() algorithm and the IKM's Feedback() algorithm are described as follows.

| Combination | Results | Interpretation | Implication |
|---|---|---|---|
| 1 | $R_1$ | NRE not used; initial injection was successful. Entry point considered vulnerable, and $R_2$ and $R_3$ are not necessary. | 1. Deterministic result (vulnerable)<br>2. NRE not used |
| 2 | $R_1 = R_2 = R_3$ | 1. All requests are filtered by validation procedure. Automated assessment is impossible.<br>2. Requests are not filtered, but Web application is not vulnerable. | 1. NRE failed<br>2. Undeterministic result |
| 3 | $R_1 = R_2 \neq R_3$ | $R_1$ and $R_2$ did not bypass validation, but $R_3$ did. | 1. Deterministic result (secure)<br>2. Feedback() called |
| 4 | $R_2 = R_3 \neq R_1$ | Malicious pattern recognized by filtering mechanism. Entry point not vulnerable. | 1. Deterministic result (secure) |
| 5 | $R_1 \neq R_2 \neq R_3$ | $R_1$ is recognized as injection pattern, $R_2$ failed validation, $R_3$ succeeded. | 1. Deterministic result (secure)<br>2. Feedback() called |

Fig. 4. The NRE for deep injection.

```
SQL_Injection()
Input: $A$—A set of arguments for the entry point.
Each argument $\bar{a} \in A = (T, V)$, where $T$ is a set of
descriptive terms (e.g., text near an input field or
the field's variable name) and $V$ is an optional
set of predefined values.
$M$—A set of malicious test patterns.
For each argument $\bar{a}_i \in A$,
  If $(|\bar{a}_i.V| > 0)$ then for each term $\hat{t} \in \bar{a}_i.T$,
  Expand_Values($\hat{t}, \bar{a}_i.V$)
  For each test pattern $\vec{m} \in M$,
    $Request1[\bar{a}_i] = \vec{m}$
    For each $\bar{a}_j \in A, \bar{a}_j \neq \bar{a}_i$,
      For each $\hat{t} \in \bar{a}_j.T$,
        $Request1[\bar{a}_j]$ =Get_Value($\hat{t}$); if $Request1$
        $[\bar{a}_j] \neq$"not found" then break;
    $R_1$ = Do_Injection($Request1$);
    If DetectErrorPatterns($R_1$) then
      ReportVulnerability($\bar{a}_i, \vec{m}$); continue
    else
      $Request2$=$Request1$;
      $Request2[\bar{a}_i]$ =$InvalidString$;
      $Request3$ = $Request1$;
      For each $\hat{t} \in \bar{a}_i.T$,
        $Request3[\bar{a}_i]$ =Get_Value($\hat{t}$); if $Request3$
        $[\bar{a}_i] \neq$"not found" then break;
      $R_2$ = Do_Injection($Request2$);
      $R_3$ = Do_Injection($Request3$)
    If $R_1$ equals $R_2$ then
      If $R_2$ equals $R_3$ then
        $Comb$ = 2 else $Comb$ = 3;
    else If $R_2$ equals $R_3$ then
      $Comb$ = 4 else $Comb$ = 5;
    Switch($Comb$)
      Case 2: ReportNREFailure();
      Case 3: Feedback(A); ReportSecure ($\bar{a}_i, \vec{m}$);
      Case 4: ReportSecure($\bar{a}_i, \vec{m}$);
      Case 5: Feedback(A); ReportSecure ($\bar{a}_i, \vec{m}$);

Feedback()
Input: $A$—A set of arguments for an entry point.
  For each argument $\bar{a}_i \in A$,
    For each $\hat{t} \in \bar{a}_j.T$,
      $\tilde{v}$ = Get_Value($\hat{t}$); if $\tilde{v} \neq$"not found" then
      break;
    $Conf(\tilde{v})$ = $Conf(\tilde{v})$ + 1
```

### 2.2.4. Injection patterns and error messages

WAVES' injection patterns are crafted not to intrude a vulnerable entry point (e.g., executing a SQL command), but to make it output database error messages. If an entry point outputs database error messages in response to a particular injection pattern, it is vulnerable to that pattern. Below are some representative injection patterns:

| | | | | | |
|---|---|---|---|---|---|
| 1: | 'waves_ scanner | 6: | 'OR '0' = '0' | 11: | 9%2c + 9%2c + 9 |
| 2: | waves_ scanner' | 7: | ') OR ('0' = '0' | 12: | "' |
| 3: | 'OR 1=1– | 8: | 'UNION' | 13: | ' |
| 4: | 'OR' | 9: | 'WHERE' | 14: | %3B |
| 5: | 'OR | 10: | '%22 | 15: | 9,9,9 |

Though Web applications have a wide selection of backend databases (e.g., SQL Server, MySQL, Oracle, Sybase, and DB2), error messages outputted by each database in response to the above injection patterns usually include a particular string. We search for such strings in an HTML output to detect database error messages. The possibility of these exact strings appearing in a normal output is low. Examples of such strings are given below:

| | |
|---|---|
| 1: | "`Database error: There was a error in executing the following SQL:`" |
| 2: | "`Microsoft OLE DB Provider for ODBC Drivers error`"; |
| 3: | "`supplied argument is not a valid MySQL result`"; |
| 4: | "`You have an error in your SQL syntax`"; |
| 5: | "`Column count doesn't match value count at row`"; |
| 6: | "`Can't find record in`"; |
| 7: | "`Incorrect column specifier for column`"; |
| 8: | "`Incorrect column name`"; |
| 9: | "`Invalid parameter type`"; |
| 10: | "`Unknown table`"; |
| 11: | "`ODBC SQL Server Driver`"; |
| 12: | "`Microsoft VBScript runtime`"; |
| 13: | "`ODBC Microsoft Access Driver`"; |

### 2.3. Cross-site scripting

As with SQL injection, cross-site scripting [16] is also associated with undesired data flow. To illuminate the basic concept, we offer the following scenario.

A Web site for selling computer-related merchandise holds a public on-line forum for discussing the newest computer products. Messages posted by users are submitted to a CGI program that inserts them into the Web application's database. When a user sends a request to view posted messages, the CGI program retrieves them from the database, generates a response page, and sends the page to the browser. In this scenario, a hacker can post messages containing malicious scripts into the forum database. When other users view the posts, the malicious scripts are delivered on behalf of the Web application [16]. Browsers enforce a Same Origin Policy [37,41] that limits scripts to accessing only those cookies that belong to the server from which the scripts were delivered. In this scenario, even though the executed script was written by a malicious hacker, it was delivered to the browser on behalf of the Web application. Such scripts can therefore be used to read the Web application's cookies and to break through its security mechanisms.

### 2.4. Cross-site scripting detection

Indications of cross-site scripting are detected during the reverse engineering phase, when a crawler performs a complete scan of every page within a Web application. Equipping a crawler with the functions of a full browser results in the execution of dynamic content on every crawled page (e.g., Javascripts, ActiveX controls, Java Applets, and Flash scripts). Any malicious script that has been injected into a Web application via cross-site scripting will attack the crawler in the same manner that it attacks a browser, thus putting our WAVES-hosting system at risk. We used the Detours [26] package to create a SEE that intercepts system calls made by a crawler. Calls with malicious parameters are rejected.

The SEE operates according to an anomaly detection model. During the initial run, it triggers a learning mode in WAVES as it crawls through predefined links that are the least likely to contain malicious code that induces abnormal behavior. Well-known and trusted pages that contain ActiveX controls, Java Applets, Flash scripts, and Javascripts are carefully chosen as crawl targets. As they are crawled, normal behavior is studied and recorded. Our results reveal that during start-up, Microsoft Internet Explorer (IE)

1. locates temporary directories,
2. writes temporary data into registry,
3. loads favorite links and history list,
4. loads the required DLL and font files,
5. creates named pipes for internal communication.

During page retrieval and rendering, IE

1. checks registry settings,
2. writes files to the user's local cache,
3. loads a cookie index if a page contains cookies,
4. loads corresponding plug-in executables if a page contains plug-in scripts.

The SEE uses the behavioral monitoring specification language (BMSL) [47,60] to record these learned normal behaviors. This design allows users to easily modify the automatically generated specifications if necessary. Fig. 5 presents an example of a SEE-generated BMSL description.

The SEE pre-compiles BMSL descriptions into a hashed policy database. During page execution and behavior stimulation, parameters of intercepted system calls are compared with this policy database. If the parameters do not match the normal behavior policy (e.g., using "C:\autoexec.bat" as a parameter to call CreateFileEx), the call is considered malicious, since IE was not monitored to make any file access under the C:\directory dur-

```
allowFile =
{"C:\WINDOWS\System32\mshtml.dll" , … (skipped) }
CreateFileW (filepath,access_mode,share_mode,SD,
             create_mode,attribute,temp_handle)
| ( filepath ∉ allowFile ) → deny
```

Fig. 5. A SEE-generated BMSL description.

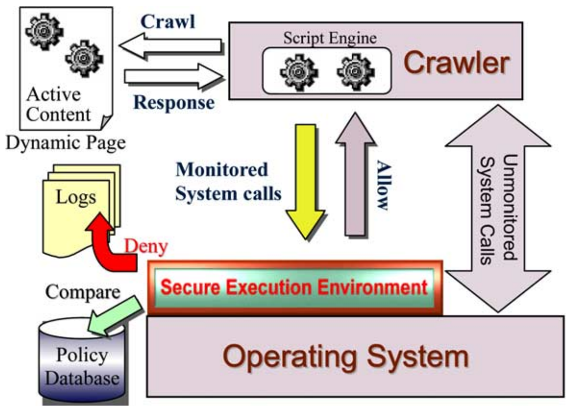


Fig. 6. The secure execution environment (SEE).

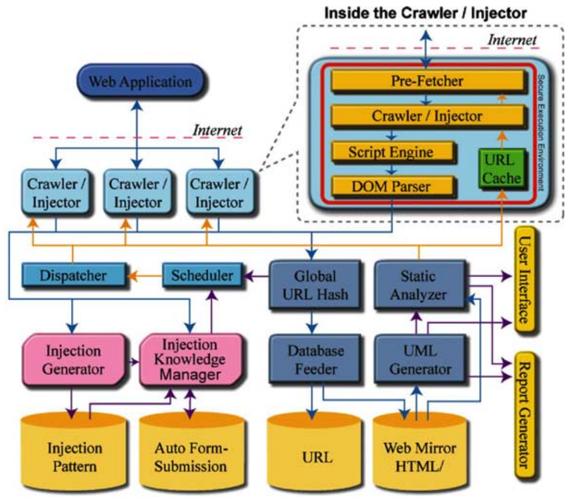


Fig. 7. System architecture of WAVES.

ing the learning phase. Fig. 6 illustrates the SEE mechanism.

The SEE provides (a) a self-protection mechanism to guard against malicious code, and (b) a method to detect malicious code inserted into Web applications. One deficiency is that the mechanism only detects code that has already been inserted, and not the weaknesses of Web applications that make them vulnerable to attack. Detecting such vulnerabilities requires an off-line static analysis of Javascripts retrieved during the reverse engineering phase. We are still in the initial phase of designing and experimenting with this analytical procedure.

## 3. System implementation

Fig. 7 depicts the entire WAVES system architecture, which we will briefly describe in this section. The crawlers 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 them exhibit the same behaviors as browsers, they were equipped with IE's DOM parser and scripting engine. We chose IE's engines over others (e.g. Gecko [39] 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 crawler execution. Please note that a subset of the above events is triggered by Web application errors. An obvious example is a Javascript error event produced by a scripting engine during a runtime interpretation of Javascript code. The crawler suppresses the dialog box that is triggered by the event, but more importantly, it logs the event and prepares corresponding entries generating an assessment report.

## 4. Related work

Offutt [42] surveyed Web managers and developers on quality process drivers and found that while time-to-market is still considered the most important quality criteria for traditional software, security is now very high on the list of concerns for Web application development. Though not specifically aimed at improving security attributes, there has been a recent burst of activity in developing methods and tools for Web application testing [8,23,51], analysis [51,55], and reverse engineering [20,21,52–54,68].

We used this paper to present a Web application security scanner—an automated software testing platform for the remote, black-box testing of

Web applications. In Bertolino's [10] words, software testing involves the "dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified behavior." Unit, regression, conformance, and other types of tests make use of carefully chosen or created test cases to identify faults. In white-box testing, a typical solution is to make test case selection based on source code—the strategy adopted by Rapps and Weyuker [50] in their data flow testing model based on program paths (which they refer to as du-paths) that connect variable definitions and uses. Many errors in variable definition remain hidden until the variable is referenced at a program point far away from its definition. Data flow criteria are used to determine the extent to which du-paths are tested. Liu et al. [34,35] were among the first to propose a model for applying data flow testing to Web applications. Treating individual Web application components as objects, they created a Web Application Test Model (WATM) to capture structural test artifacts. According to du-paths of interest, test cases are derived from flow graphs generated via an analysis of a Web application's source code.

Andrews, Offutt and Alexander proposed a system-level testing technique based on finite state machines (FSMs) that model subsystems of Web applications. Test requirements are generated as FSM state subsequences, which are then combined to form complete executable tests. In general, all the above mentioned Web application testing approaches are "white-box" approaches that first build system models from source code and then use the models to identify test requirements. Generating test cases for the requirements require extensive user participation. In contrast, WAVES is a WSS that performs remote, black-box testing with automated test case generation (see WSS definition in Section 2.2).

In reverse engineering a Web application, WAVES uses what we call a "complete crawling" mechanism to attempt more complete crawls. This is accomplished by three strategies—browser emulation, user event generation, and automated form completion. Similar efforts were made for the VeriWeb [8] project, which addresses the automated testing of dynamic Web applications. VeriWeb embeds Gecko [39] for browser emulation, while WAVES embeds IE. IE was our first choice because most browser attacks are aimed at IE instead of Netscape Navigator. Both VeriWeb and WAVES perform automated form submissions, a reflection of studies on searching the hidden Web [9,27,33,48]. To automatically generate valid input data, VeriWeb uses Smart Profiles, which represents sets of user-specified attribute-value pairs. In contrast, WAVES incorporates a self-learning knowledge base.

Scott and Sharp [58] take a different approach to protecting against SQL injection and cross-site scripting attacks: a global input validation mechanism. They argue that Web application vulnerabilities are essentially unavoidable, meaning that security assurance needs to be "abstracted" to a higher level. However, to adapt this mechanism to a legacy Web application requires that rules be defined for every single data entry point—perhaps a difficult task for Web applications that have been developed over a long time period, since they often contain complicated structures with little documentation. It would be unusual for a Web manager to be familiar with all of the data entry points for a site with thousands of pages. *AppShield* [56] and *InterDo* [30] are two commercial products similar to Scott and Sharp's application-level firewall. However, experience reports suggest that making these firewalls work requires careful configuration by experienced security professionals [12].

Another protection approach, the ⟨bigwig⟩ project [15], also provides Web application input validation mechanisms. The mechanism is designed to automatically generate server- and client-side validation routines. However, it only works with Web applications developed with the ⟨bigwig⟩ language. In contrast, WAVES provides security assurance without requiring modifications to existing Web application architectures. The Open Web Application Security Project (OWASP) [44] has launched a WebScarab [44] project aimed at developing a security assessment tool very similar to WAVES. Sanctum has recently incorporated routines to detect SQL injection vulnerabilities in Web applications into its *AppScan* [57]. Two other available commercial scanners include SPI Dynamics' *Web-*

*Inspect* [66] and Kavado's *ScanDo* [30]. Reviews of these tools can be found in [3].

Similar to WAVES, our other project *WebSSARI* [24,25] also attempts to provide automated Web application security assessment. WAVES is a black-box testing framework, while WebSSARI is a white-box verification tool. WAVES can assess up and running applications, giving it two advantages: (a) assessment is provided without source code, and (b) the actual behaviors of the running application is tested and observed. A side effect is that such testing may cause permanent state changes to the target application. WebSSARI requires the application's source code, which is used to simulate (or predict) the application's runtime behaviors. Unlike testing, which cannot guarantee coverage, verification guarantees coverage, and therefore, soundness. In other words, WAVES tries its best to discover bugs in up and running applications, while WebSSARI tries to guarantee the absence of bugs from their source code. A drawback of WebSSARI is that it verifies against simulated runtime behaviors based on program abstraction, so the results are only as correct as the abstraction and simulation. On the other hand, a testing framework always observes the actual behaviors of a running application.

To expedite the reverse engineering and fault injection processes, the multi-threaded WAVES crawler performs parallel crawls. We adopted many of the ideas and strategies reviewed in [17,64] to construct fast, parallel crawlers. For the automated form completion task, we followed suggestions offered by Bergman [9] and Raghavan [48], but incorporated a more complex self-learning knowledge base.

Behavior monitoring has attracted research attention due to its potential to protect against unknown or polymorphic viruses [6,7,11,14]. In addition to self-protection, we used behavior monitoring to detect malicious content before it reaches users. Furthermore, WAVES performs behavior stimulation to induce malicious behavior in the monitored components. In other words, it uses behavior monitoring for both reactive and proactive purposes.

We employed sandboxing technology to construct a self-contained SEE. Our SEE implementation is based on descriptions in [26,29,31]. In [29], a generic model is proposed for sandboxing downloaded components. Regarding the actual implementation, we had a choice between two open-source toolkits—Detours [26] and GSWTK [31]. We selected Detours because of its lighter weight. For a standard description of normal behaviors, we used BMSL [47,60]. We compared our SEE with other commercial sandboxes, including Finjan's *SurfinShield* [22], Aladin's *ESafe* [1], and Pelican's *SafTnet* [45,46]. Surveys of commercially available sandboxes can be found in [2,71].

## 5. Experimental results

We tested for thoroughness by comparing the number of pages retrieved by various crawlers. Teleport [67] proved to be the most thorough of a group of crawlers that included WebSphinx [38], Larbin [61], and Web-Glimpse [36]. 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.

To test WAVES' application in real-world situations, we selected 14 well-known sites that reflect a variety in size, nature, and design. On average, WAVES retrieved 28% more pages than Teleport when tested with the 14 sites (Fig. 8). We attribute the discovery of the extra pages to WAVES' script interpretation and automated form completion capabilities.

| Site | Forms of Interest | Waves | Teleport | WAVES' Advantage |
|---|---|---|---|---|
| www.nai.com | 52 | 14,589 | 11,562 | 21% |
| www.lucent.com | 21 | 8,198 | 7,929 | 3% |
| www.trendmicro.com | 70 | 5,781 | 2,939 | 49% |
| www.palm.com | 43 | 4,459 | 4,531 | -2% |
| www.olympic.org | 9 | 4,389 | 3,069 | 30% |
| www.apache.org | 5 | 3,598 | 3,062 | 15% |
| www.verisign.com | 42 | 3,231 | 3,069 | 5% |
| www.ulead.com | 3 | 1,624 | 1,417 | 13% |
| www.cert.org | 3 | 1,435 | 1,267 | 12% |
| www.maxtor.com | 4 | 1,259 | 863 | 31% |
| www.mazda.com | 1 | 1,030 | 356 | 65% |
| www.linuxjournal.com | 7 | 871 | 167 | 81% |
| www.cadillac.com | 2 | 673 | 598 | 11% |
| www.web500.com | 3 | 564 | 237 | 58% |

Fig. 8. Crawling statistics for WAVES and Teleport.

| Site | P(S) | P(C\|S) | P($C_L$\|S) | P(N) | P($F_0$\|V,D) | P($F_L$\|V,D) | P($F_{LN}$\|V,D) |
|---|---|---|---|---|---|---|---|
| NAI | 18.69 | 80.32 | 81.93 | 70.58 | 19.68 | 18.07 | 05.31 |
| Lucent | 83.90 | 79.87 | 83.76 | 77.70 | 20.13 | 16.24 | 03.62 |
| Trend Micro | 90.72 | 78.52 | 84.04 | 98.60 | 21.48 | 15.96 | 00.22 |
| Palm | 43.56 | 88.63 | 92.20 | 100 | 11.37 | 07.80 | 0 |
| Olympic | 88.23 | 100 | 100 | 100 | 0 | 0 | 0 |
| Apache | 75.00 | 77.77 | 77.77 | 100 | 22.23 | 22.23 | 22.23 |
| Verisign | 89.93 | 86.06 | 95.27 | 93.02 | 13.94 | 04.73 | 00.33 |
| Ulead | 100 | 83.72 | 91.86 | 100 | 16.28 | 08.14 | 0 |
| Cert | 55.55 | 100 | 100 | 100 | 0 | 0 | 0 |
| Maxtor | 96.77 | 36.66 | 51.66 | 100 | 63.34 | 48.34 | 0 |
| Mazda | 100 | 100 | 100 | 100 | 0 | 0 | 0 |
| Linux Journal | 100 | 84.61 | 84.61 | 100 | 15.39 | 15.39 | 0 |
| Cadillac | 100 | 73.30 | 86.60 | 25.00 | 26.70 | 13.40 | 10.05 |
| Web500 | 91.30 | 67.80 | 79.50 | 100 | 32.20 | 20.50 | 0 |
| Average | 80.93 | 81.13 | 86.06 | 90.99 | 18.76 | 13.62 | 02.46 |

Fig. 9. Automated submission results.

To test the injection algorithm, WAVES was configured to identify all forms of interest (i.e., those containing textboxes; see column 2 of Fig. 8), to perform an NRE for each form, to fill in and submit the form, and to make a judgment on submission success based on the reply page and the previously retrieved NRE. WAVES creates detailed logs of the data used for each automated form completion, the resulting HTML page, and submission success judgments. In Fig. 9 (produced from a manual inspection of the logs), P(S) denotes the probability that the semantics of an input textbox have been successfully extracted; $P(C|S)$ denotes the conditional probability that a form completion is successful given that semantic extraction was successful; $P(C_L|S)$ denotes the same probability, but after a learning process in which the IKM expands its knowledge base; P(N) denotes the probability of a successful NRE process; and $P(F|V,D)$ denotes the probability of false negatives given that a form is both validated and defected (i.e., vulnerable). False negatives are induced when all of the following conditions are true: (a) the form is defected (vulnerable); (b) the form is validated; (c) WAVES cannot correctly complete the form; and (d) the NRE process fails, but WAVES is unable to recognize the failure. One common example of (d) is when combination 5 (see Fig. 4) occurs, but the differences in $R_1$, $R_2$ and $R_3$ do not result because $R_1$ is recognized as malicious, $R_2$ recognized as invalid, and $R_3$ considered valid. Instead, all requests are recognized as invalid (because WAVES in incapable of supplying valid values), and the differences in the replies result simply from random advertising content inserted into each response HTML page. Therefore, although NRE considers this entry point secure, it may be actually vulnerable to a human hacker capable of supplying a combination of malicious and valid values.

Based on the four conditions, a general definition of the probability of false negatives given that the form being tested enforces validation can be defined as $P(F|V,D) = (1 - P(C|S) - P(C|X))*(1 - P(U|N))$, where $P(C|X)$ denotes the probability that form completion has succeeded given that semantic extraction failed, and $P(U|N)$ denotes the probability that WAVES is unable to determine an NRE failure. $P(U|N)$ depends large on a website's output and layout style. It can be improved by replacing the current simple page similarity algorithm (used to differentiate HTML response pages) with one that (a) considers block importance within an HTML page and (b) makes comparisons based on the most important blocks. Song et al. [65] claims an 85.9% accuracy for their Micro-Accuracy algorithm for identifying important HTML blocks. We consider it our future work to incorporate their algorithm into WAVES.

In our analysis, we used the pessimistic definition of $P(C|X) = 0$, meaning that we assumed zero probability of correctly filling a validated form whose semantics could not be extracted.

As a part of our approach, both self-learning (to assist automated submissions) and NRE are used in order to decrease false negative rates when injecting a validated form. To evaluate these mechanisms, we define three probabilities derived from $P(F|V,D)$: $P(F_0|V,D)$, $P(F_L|V,D)$, and $P(F_{LN}|V,D)$. $P(F_0|V,D)$ denotes the probability of $P(F|V,D)$ when neither the self-learning nor the NRE algorithms are applied. $P(F_L|V,D)$ denotes the probability of $P(F|V,D)$ when the learning mode is enabled. $P(F_{LN}|V,D)$ denotes the probability when applying both learning and NRE. As Fig. 9 shows, the $P(F|V,D)$ average decreased more than 5% (from the 18.76% of $P(F_0|V,D)$ to the 13.62% of $P(F_L|V,D)$) in other words, during this experiment, the WAVES' learning mechanism decreased the rate of false negatives by 5%. An additional drop of 11% occurred between $P(F_L|V,D)$ and $P(F_{LN}| V,D)$ due to a contribution from the NRE algorithm. In total, WAVES' self-learning knowledge base and the NRE algorithm combined contributed to a 16% decrease in false negatives, to a final rate of 2.46%.

Among the 14 sites, Olympic, Cert and Mazda had the highest $P(C| S)$ and $P(C_L|S)$ rate—100%. After examining their HTML pages, we concluded that this resulted from the website developers stuck solidly by good coding practices, giving each variable a meaningful name. Maxtor had both the lowest $P(C|S)$ and the largest increase from $P(C|S)$ to $P(C_L|S)$, meaning that IKM's learning proved most helpful for the site. After manual examination, we found that in some of its pages, Maxtor asks users to enter a hard disk model number, while in some it offers an option list allowing users to make a direct selection. Learning allows WAVES to supply valid values (model numbers) that it would otherwise not know. The same reason explains the large differences exhibited also by Verisign, Ulead, and Cadillac.

In order to use behavior monitoring for malicious script detection, the WAVES crawler was modified to accommodate IE version 5.5 instead of 6.0 because of the greater vulnerability of the older version. To incorporate the most recent version would mean that we could only detect new and unknown forms of attacks. Furthermore, the behavior monitoring process is also dependent upon the crawler's ability to simulate user-generated events as test cases, and IE versions older than 5.5 do not support event simulation functions.

SecurityGlobal.net classified the impacts of vulnerabilities discovered between April, 2001 and March, 2002 into 16 categories [62]. We believe the items on this list can be grouped into four general categories: (1) restricted resource access, (2) arbitrary command execution, (3) private information disclosure, and (4) denial of service (DoS). We gathered 26 working exploits that demonstrated impacts associated with the first three categories, and used them to create a site to test our behavior monitoring mechanism. For this test, WAVES was installed into an unpatched version of Windows 2000. Fig. 10 lists the impact categories and observed detection ratios.

WAVES successfully detected category 1 and 2 impacts. One reason for this high accuracy rate is that IE exhibited very regular behavior during the normal-behavior learning phase. The system calls that IE makes are fixed, as are the directories and files that it accesses; such clearly defined behavior makes it easier to detect malicious behavior. Our exploits that demonstrate category 3 impacts operate by taking advantage of certain design flaws of IE. By tricking IE into misinterpreting the origins of Javascripts, these exploits break the Same Origin Policy [37,41] enforced by IE and steals user cookies. Since these design flaws leak application-specific data, they are more transparent to a SEE and are therefore more difficult to detect. This is reflected in our test using three commercial sandboxes—*SurfinShield* [22], *ESafe* [1], and *SafTnet* [45,46]. Similar to WAVES, none of the sandboxes was able to detect any of the six exploits of Category 3. As well as for impacts of Category 4, a more sophisticated mechanism must be implemented for detection, and is an area of our future research.

The SEE does not intercept all system calls. Doing so may allow the SEE to gather more

| Class of Impact | Exploits | Detection Ratio |
|---|---|---|
| 1) Restricted resource access | 9 | 9/9 |
| 2) Arbitrary command execution | 9 | 9/9 |
| 3) Private information disclosure | 6 | 0/6 |
| 4) Denial of service (DOS) | 2 | 0/2 |

Fig. 10. Detection ratios for each class of impact.

| File Management | Process Management |
|---|---|
| CreateFile | CreateProcess |
| WriteFile | CreateProcessAsUser |
| CreateFileMapping | CreateProcessWithLogonW |
| Directory Management | OpenProcess |
| CreateDirectory | TerminateProcess |
| RemoveDirectory | Communication |
| SetCurrentDirectory | CreatePipe |
| Hook | CreateProcessWithLogonW |
| SetWindowsHookEx | Registry Access |
| System Information | RegSetValueEx |
| GetComputerName | RegOpenKeyEx |
| GetSystemDirectory | RegQueryValueEx |
| GetSystemInfo | User Profiles |
| GetSystemMetrics | GetAllUsersProfileDirectory |
| GetSystemWindowsDirectory | LoadUserProfile |
| GetUserName | GetProfilesDirectory |
| GetVersion | Windows Networking |
| GetWindowsDirectory | WNetGetConnection |
| SetComputerName | Socket |
| SystemParametersInfo | Bind |
|  | Listen |

Fig. 11. System calls intercepted by the SEE.

information, but will also induce unacceptable overhead. Therefore, the set of intercepted system calls was carefully chosen to contain calls that IE does not normally make, but that malicious components needs to make. A list of intercepted system calls is given in Fig. 11. The Detours interception module has a maximum penalty of 77 clock cycles per intercepted system call. Even for a slow CPU such as the Pentium Pro, this only amounts to approximately 15 μs. Since IE does not call most intercepted calls after initialization, the interception mechanism costs little in terms of overhead. Greater overhead potential lies in the policy matching process that determines whether a call is legal by looking at its parameters. The regular behavior exhibited by IE resulted in only 33 rules being generated by the learning process. Since the rules (expressed in BMSL) are pre-compiled and stored in memory using a simple hash, matching call parameters against these rules cost little in terms of overhead.

In addition, the event generation process was inexpensive in terms of CPU cost. Our experimental scans show that the index page of http://www.lucent.com/ contained 635 DOM elements, 126 of which carried the onMouseOver event handler. In other words, the 126 elements execute a block of pre-assigned code whenever the user

```
For i:=1 to TotalElements do Begin
  If Assigned(Elements[i].onmouseover) then do Begin
    Event = Doc.CreateEvent();
    Doc.FireEvent(Elements[i], “onmouseover”, Event);End;
End;
```

Fig. 12. Our event-generation routine.

moves a mouse over the elements. The routine used to generate the onMouseOver event for all 126 elements is shown in Fig. 12. For a 2 GHz Pentium IV, this routine took approximately 300 ms.

Thus, we conclude that while successfully intercepting malicious code of category 1 and 2, the behavior monitoring mechanism was cost-effective and feasible. However, as more sophisticated strategies are used to detect category 3 and 4 impacts, larger overheads may be induced. Note that the event-generation routine contributes not only to behavior monitoring, but also to a more complete URL discovery.

## 6. Conclusion

Our proposed mechanisms for assessing Web application security were constructed from a software engineering approach. We designed a crawler interface that incorporates and mimics Web browser functions in order to test Web applications using real-world scenarios. During the first assessment phase, the crawler attempts to perform a complete reverse engineering process to identify all data entry points—possible points of attack–of a Web application. These entry points then serve as targets for a fault injection process in which malicious patterns are used to determine the most vulnerable points. We also proposed the NRE algorithm to eliminate false negatives and to allow for “deep injection.” In “deep injection”, the IKM formulates an invalid input pattern to retrieve a negative response page, then uses an automated form completion algorithm to formulate the most likely injection patterns. After sending the injection, WAVES analyzes the resulting pages using the NRE algorithm, which is simpler, yet more accurate than the LITE approach [48]. A summary of our contributions is presented in Fig. 13.

One contribution is an automated form submission algorithm that is used by both the crawler and

| Mechanisms | Based on | Facilitates |
|---|---|---|
| Self-learning knowledge base | Topic Model | 1. Complete crawling<br>2. Deep injection |
| Negative response extraction (NRE) | Page similarity | Deep injection |
| Intelligent form parser | DOM object locality | Deep injection |
| Complete crawling | 1.Javascript engine<br>2.DOM parser<br>3.Javascript event generation | Web application Testing interface |
| Behavior monitoring | 1.Self-training, anomaly detection model<br>2.Event simulation (test case generation)<br>3. Detours (sandboxing) | 1. Self-protection<br>2. Cross-site scripting detection<br>3. Unknown malicious script detection |
| Behavior stimulation | Event simulation<br>(Test case generation) | 1. Behavior monitoring<br>2. Complete crawling |

Fig. 13. A summary of our contributions.

IKM. Here we propose two strategies to assist this algorithm. To extract the semantics of a form's input fields, we designed an "intelligent form parser" (similar to the one used in LITE [48]) that uses DOM object locality information to assist in automated form completion. However, our implementation is enhanced by incorporating a fully-functional DOM parser, as opposed to an approximate DOM parser used in [48]. To automatically provide semantically correct values for a form field, we propose a self-learning knowledge base based on the Topics model.

Finally, we added a secure execution environment (SEE) to the crawler in order to detect malicious scripts by means of behavior monitoring. The crawler simulates user-generated events as test cases to produce more comprehensive behavior observations—a process that also aids in terms of crawl thoroughness. While functioning as a self-protection mechanism, the SEE also allows for the detection of both known and unknown malicious scripts.

As a testing platform, WAVES provides the following functions, most of which are commonly required for Web application security tests:

1. Identifying data entry points.
2. Extracting the syntax and semantics of an input field.
3. Generating potentially valid data for an input field.
4. Injecting malicious patterns on a selected input field.
5. Formatting and sending HTTP requests.
6. Analyzing HTTP replies.
7. Monitoring a browser's behavior as it executes active content delivered by a Web application.

As an interface between testing techniques and Web applications, WAVES can be used to conduct a wide variety of vulnerability tests, including cookie poisoning, parameter tampering, hidden field manipulation, input buffer overflow, session hijacking, and server misconfiguration—all of which would otherwise be difficult and time-consuming tasks.

## References

[1] Aladdin Knowledge Systems, eSafe Proactive Content Security, Available from: <http://www.ealaddin.com/>.

[2] I. Armstrong, Mobile code stakes its claim, SC Magazine, Cover Story, November 2000.

[3] L. Auronen, Tool-based approach to assessing Web application security, Helsinki University of Technology, 2002.

[4] W3C, Document Object Model (DOM). Available from: <http://www.w3.org/DOM/>.

[5] C. Anley, Advanced SQL Injection in SQL Server Applications, An NGS Software Insight Security Research (NISR) Publication, 2002.

[6] F. Apap, A. Honig, S. Hershkop, E. Eskin, S. Stolfo, Detecting malicious software by monitoring anomalous windows registry accesses, in: Fifth International

Symposium on Recent Advances in Intrusion Detection, Zurich, Switzerland, October 2002.

[7] R. Balzer, Assuring the safety of opening email attachments, in: DARPA Information Survivability Conference & Exposition II, 2, 2001, pp. 257–262.

[8] M. Benedikt, J. Freire, P. Godefroid, VeriWeb: Automatically testing dynamic web sites, in: Proceedings of the 11th International Conference on the World Wide Web, Honolulu, Hawaii, May 2002.

[9] M.K. Bergman, The deep Web: surfacing hidden value, Deep Content Whitepaper, 2001.

[10] A. Bertolino, Knowledge area description of software testing, in: Guide to the Software Engineering Body of Knowledge SWEBOK (v. 0.7), Chapter 5, Software Engineering Coordinated Committee (Joint IEEE Computer Society-ACM Committee), April, 2000. Available from: <http://www.swebok.org>.

[11] M. Bernaschi, E. Gabrielli, L.V. Mancini, Operating system enhancements to prevent the misuse of system calls, in: Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, 2000.

[12] M. Bobbitt, Bulletproof web security, Network Security Magazine, TechTarget Storage Media, May 2002. Available from: <http://infosecuritymag.techtarget.com/2002/may/bulletproof.shtml>.

[13] C.M. Bowman, P. Danzig, D. Hardy, U. Manber, M. Schwartz, D. Wessels, Harvest: a scalablecustomizable discovery and access system, Department of Computer Science, University of Colorado, Boulder, 1995.

[14] T. Bowen, M. Segal, R. Sekar, On preventing intrusions by process behavior monitoring, in: Eighth USENIX Security Symposium, Washington, DC, August 1999.

[15] C. Brabrand, A.M.I. Møller, The ⟨bigwig⟩ project, ACM Transactions on Internet Technology 2 (2) (2002) 79–114.

[16] CERT, CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. Available from: <http://www.cgisecurity.com/articles/xss-faq.shtml>.

[17] J. Cho, H. Garcia-Molina, Parallel crawlers, in: Proceedings of the 11th International Conference on the World Wide Web, Honolulu, Hawaii, May 2002, pp. 124–135.

[18] M. Curphey, D. Endler, W. Hau, S. Taylor, T. Smith, A. Russell, G. McKenna, R. Parke, K. McLaughlin, N. Tranter, A. Klien, D. Groves, I. By-Gad, S. Huseby, M. Eizner, R. McNamara, A guide to building secure Web applications, The Open Web Application Security Project v.1.1.1, September 2002.

[19] DHTML Central, HierMenus. Available from: <http://www.webreference.com/dhtml/hiermenus/>.

[20] Di Lucca, G.A. Di Penta, M. Antoniol, G. Casazza, An approach for reverse engineering of Web-based applications, in: Proceedings of the Eighth Working Conference on Reverse Engineering, Stuttgart, Germany, October 2001, pp. 231–240.

[21] G.A. Di Lucca, A.R. Fasolino, F. Pace, P. Tramontana, U. De Carlini, WARE: A tool for the reverse engineering of web applications, in: Proceedings of the Sixth European Conference on Software Maintenance and Reengineering, Budapest, Hungary, March 2002, pp. 241–250.

[22] Finjan Software, Your window of vulnerability – why anti-virus isn't enough. Available from: <http://www.finjan.com/mcrc/overview.cfm>.

[23] R. Gold, HttpUnit. Available from: <http://httpunit.sourceforge.net/>.

[24] Y.W. Huang, S.K. Huang, T.P. Lin, C.H. Tsai, Securing Web application code by static analysis and runtime protection, in: Proceedings of the 13th International World Wide Web Conference, New York, May 17–22, 2004.

[25] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, S.Y. Kuo, Verifying Web applications using bounded model checking, in: Proceedings of the 2004 International Conference Dependable Systems and Networks (DSN2004), Florence, Italy, June 28–July 1, 2004.

[26] G. Hunt, D. Brubacher, Detours: binary interception of Win32 functions, in: USENIX Technical Program—Windows NT Symposium 99, 1999.

[27] P. Ipeirotis, L. Gravano, Distributed search over the hidden Web: hierarchical database sampling and selection, in: The 28th International Conference on Very Large Databases, Hong Kong, China, August 2002, pp. 394–405.

[28] J. Joshi, W. Aref, A. Ghafoor, E. Spafford, Security models for Web-based applications, Communications of the ACM 44 (2) (2001) 38–44.

[29] H. Kaiya, K. Kaijiri, Specifying runtime environments and functionalities of downloadable components under the sandbox model, in: Proceedings of the International Symposium on Principles of Software Evolution, Kanazawa, Japan, November 2000, pp. 138–142.

[30] Kavado, Inc., InterDo Version 3.0., Kavado Whitepaper, 2003.

[31] C. Ko, T. Fraser, L. Badger, D. Kilpatrick, Detecting and countering system intrusions using software wrappers, in: Proceedings of the 9th USENIX Security Symposium, Denver, Colorado, August 2000.

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

[33] S. Liddle, D. Embley, D. Scott, S.H. Yau, Extracting data behind web forms, in: Proceedings of the Workshop on Conceptual Modeling Approaches for e-Business, Tampere, Finland, October 2002.

[34] C.H. Liu, D.C. Kung, P. Hsia, C.T. Hsu, Structural testing of Web applications, in: Proceedings of the 11th International Symposium Software Reliability Engineering (ISSRE2000), October 8–11, 2000, pp. 84–96.

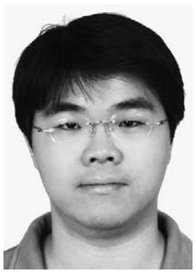
[35] C.H. Liu, D.C. Kung, P. Hsia, C.T. Hsu, Object-based data flow testing of Web applications, in: Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS'00), Hong Kong, China, October 30–31, 2000.

[36] U. Manber, M. Smith, B. Gopal, WebGlimpse—combining browsing and searching, in: Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, California, January 1997).

[37] Microsoft, Scriptlet Security, Getting Started with Scriptlets, MSDN Library, 1997. Available from: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnindhtm/html/instantdhtmlscriptlets.asp>.
[38] R.C. Miller, K. Bharat, SPHinx: a framework for creating personal, site-specific Web crawlers, in: Proceedings of the 7th International World Wide Web Conference, Brisbane, Australia, April 1998, pp. 119–130.
[39] Mozilla.org, Mozilla Layout Engine. Available from: <http://www.mozilla.org/newlayout/>.
[40] P.G. Neumann, Risks to the public in computers and related systems, ACM SIGSOFT Software Engineering Notes 25 (3) (2000) 15–23.
[41] Netscape, JavaScript Security in Communicator 4.x. Available from: <http://developer.netscape.com/docs/manuals/communicator/jssec/contents.htm#1023448>.
[42] J. Offutt, Quality attributes of web software applications, IEEE Software 19 (2) (2002) 25–32.
[43] K. Ohmaki, Open source software research activities in aist towards secure open systems, in: Proceedings of the 7th IEEE International Symposium High Assurance Systems Engineering (HASE'02), Tokyo, Japan, October 23–25, 2002, p. 37.
[44] OWASP, WebScarab Project. Available from: <http://www.owasp.org/webscarab/>.
[45] Pelican Security Inc., Active content security: risks and solutions, Pelican Security Whitepaper, 1999.
[46] P. Privateer, Making the net safe for ebusiness: solving the problem of malicious Internet mobile code, in: Proceedings of the eSolutions World 2000 Conference, Philiadelphia, Pennsylvania, September 2000.
[47] P. Uppuluri, R. Sekar, Experiences with specification based intrusion detection system, in: Fourth International Symposium on Recent Advances in Intrusion Detection, Davis, California, October 2001.
[48] S. Raghavan, H. Garcia-Molina, Crawling the hidden Web, in: Proceedings of the 27th VLDB Conference, Roma, Italy, September 2001, pp. 129–138.
[49] S. Raghavan, H. Garcia-Molina, Crawling the hidden Web, in: Technical Report 2000–36, Database Group, Computer Science Department, Stanford, November 2000.
[50] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, IEEE Transactions on Software Engineering SE-11 (1985) 367–375.
[51] F. Ricca, P. Tonella, Analysis and testing of Web applications, in: Proceedings of the 23rd IEEE International Conference on Software Engineering, Toronto, Ontario, Canada, May 2001, pp. 25–34.
[52] F. Ricca, P. Tonella, I.D. Baxter, Restructuring Web applications via transformation rules, Information and Software Technology 44 (13) (2002) 811–825.
[53] F. Ricca, P. Tonella, Understanding and restructuring Web sites with ReWeb, IEEE Multimedia 8 (2) (2001) 40–51.
[54] F. Ricca, P. Tonella, Web application slicing, in: Proceedings of the IEEE International Conference on Software Maintenance, Florence, Italy, November 2001, pp. 148–157.
[55] F. Ricca, P. Tonella, Web site analysis: structure and evolution, in: Proceedings of the IEEE International Conference on Software Maintenance, San Jose, California, October 2000, pp. 76–86.
[56] Sanctum Inc., AppShield 4.0 Whitepaper, 2002. Available from: <http://www.sanctuminc.com>.
[57] Sanctum Inc., Web Application Security Testing—AppScan3.5. Available from: <http://www.sanctuminc.com>.
[58] D. Scott, R. Sharp, Abstracting application-level Web security, in: The 11th International Conference on the World Wide Web, Honolulu, Hawaii, May 2002, pp. 396–407.
[59] D. Scott, R. Sharp, Developing secure Web applications, IEEE Internet Computing 6 (6) (2002) 38–45.
[60] R. Sekar, P. Uppuluri, Synthesizing fast intrusion detection/prevention systems from high-level specifications, in: USENIX Security Symposium, 1999.
[61] Sebastien@ailleret.com, Larbin: A multi-purpose Web crawler. Available from: <http://larbin.sourceforge.net/index-eng.html>.
[62] SecurityGlobal.net, Security Tracker Statistics, April 2002–March 2002. Available from: <http://securitytracker.com/learn/statistics.html>.
[63] U. Shankar, K. Talwar, J.S. Foster, D. Wagner, Detecting format string vulnerabilities with type qualifiers, in: Proceeding of the 10th USENIX Security Symposium (USENIX'02), Washington DC, August 2002, pp. 201–220.
[64] V. Shkapenyuk, T. Suel, Design and implementation of a high-performance distributed Web crawler, in: Proceedings of the 18th IEEE International Conference on Data Engineering, San Jose, California, Febraury 2002, pp. 357–368.
[65] R. Song, H. Liu, J.R. Wen, W.Y. Ma, Learning block importance models for Web pages, in: Proceedings of the 13th International World Wide Web Conference, New York, May 17–22, 2004, pp. 203–211.
[66] SPI Dynamics, Web application security assessment, SPI Dynamics Whitepaper, 2003.
[67] Tennyson Maxwell Information Systems, Inc., Teleport Webspiders. Available from: <http://www.tenmax.com/teleport/home.htm>.
[68] S. Tilley, S. Huang, Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: A case study, in: Proceedings of the 23rd IEEE International Conference on Software Engineering, Toronto, Ontario, Canada, May 2001, pp. 514–523.
[69] United States Patent and Trademark Office. Available from: <http://www.uspto.gov/patft/>.
[70] S. Varghese, Microsoft patches critical Hotmail hole, TheAge.com, March 24, 2004. Available from: <http://www.theage.com.au/articles/2004/03/24/1079939690076.html>.
[71] R. Vibert, AV alternatives: extending scanner range, in: Information Security Magazine, February 2001.
[72] J. Voas, G. McGraw, Software Fault Injection: Inoculating Programs against Errors, Wiley, New York, 1997, pp. 47–48.
[73] WinMerge, WinMerge: A visual text file differencing and merging tool for Win32 platforms. Available from: <http://winmerge.sourceforge.net>.

**Yao-Wen (Wayne) Huang** received his B.S. (1996) and M.S. (1998) degrees in Computer Science and Information Engineering from National Chiao Tung University. In addition to serving as a research assistant at the Institute of Information Science of Academia Sinica, he is working toward his Doctorate in Electrical Engineering at National Taiwan University. His research interests include software quality, formal methods, information security, and fault-tolerant designs.

**Chung-Hung Tsai** earned his M.S. in Computer Science and Information Engineering from National Chiao-Tung University in 2002. He worked as a research assistant at the Institute of Information Science, Academia Sinica, while pursuing his master's degree. He is currently serving his mandatory military service as an embedded software designer. His research interests include software quality, information security, and distributed systems.



**Tsung-Po Lin** received his B.S. from Tunghai University in 1998 and his M.S. from National Chiao Tung University in 2000—in both cases, in Computer Science and Information Engineering. Since then he has been working at Academia Sinica's Institute of Information Science as a research assistant. His research interests include network security, system security, and digital rights management.

**Shih-Kun Huang** is a faculty member in the Department of Computer Science and Information Engineering at National Chiao Tung University in Hsinchu, Taiwan and jointly with the Institute of Information Science, Academia Sinica. His research interests are in open source software engineering, object-oriented technology and software quality. He received his B. S., M. S. and Ph.D. degrees in Computer Science and Information Engineering from National Chiao Tung University in 1989, 1991 and 1996 respectively.

**D.T. Lee** received his B.S. degree in Electrical Engineering from the National Taiwan University in 1971, and the M.S. and Ph. D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1976 and 1978 respectively. Dr. Lee has been with the Institute of Information Science, Academia Sinica, Taiwan, where he is a Distinguished Research Fellow and Director since July 1, 1998. Prior to joining the Institute, he was a Professor of the Department of Electrical and Computer Engineering, Northwestern University, where he has worked since 1978. His research interests include design and analysis of algorithms, computational geometry, VLSI layout, web-based computing, algorithm visualization, software security, bio-informatics, digital libraries and advanced IT for intelligent transportation systems. He has published over 120 technical articles in scientific journals and conference proceedings, and he also holds three U.S. patents, and one R.O.C patent. He is Editor of Algorithmica, Computational Geometry: Theory & Applications, ACM Journal of Experimental Algorithmics, International Journal of Computational Geometry & Applications, Journal of Information Science and Engineering, and Series Editor of Lecture Notes Series on Computing for World Scientific Publishing Co., Singapore. He is Fellow of IEEE, Fellow of ACM, President of IICM and Academician of Academia Sinica.

**Sy-Yen Kuo** is a Professor at the Department of Electrical Engineering, National Taiwan University and was the Chairman from 2001 to 2004. Currently he is on sabbatical leave and serving as a Visiting Professor at the Computer Science and Engineering Department, the Chinese University of Hong Kong. He received the BS (1979) in Electrical Engineering from National Taiwan University, the MS (1982) in Electrical & Computer Engineering from the University of California at Santa Barbara, and the PhD (1987) in Computer Science from the University of Illinois at Urbana-Champaign. He spent his sabbatical year as a visiting researcher at AT&T Labs-Research, New Jersey from 1999 to 2000. He was the Chairman of the Department of Computer Science and Information Engineering, National Dong Hwa University, Taiwan from 1995 to 1998, a faculty member in the Department of Electrical and Computer Engineering at the University of Arizona from 1988 to 1991, and an engineer at Fairchild Semiconductor and Silvar-Lisco, both in California, from 1982 to 1984. In 1989, he also worked as a summer faculty fellow at Jet Propulsion Laboratory of California Institute of Technology. His current research interests include dependable systems and networks, software reliability engineering, mobile computing, and reliable sensor networks. Prof. Kuo is the Steering Committee Chair of the Pacific Rim International Symposium on Dependable Computing (PRDC) and the Associate Editor of the IEEE Trans. on Dependable and Secure Computing. He serves/served as a member of the Board of Directors and officers in the IEEE Taipei Section. Professor Kuo is an IEEE Fellow for his contributions on dependable computing and software reliability engineering. He has published more than 230 papers in journals and conferences. He received the distinguished research award (1997-2005) from the National Science Council, Taiwan. He was nominated for Best Paper Award in the 13th International World Wide Web Conference (WWW2004) and also a recipient of the Best Paper Award in the 1996 International Symposium on Software Reliability Engineering, the Best Paper Award in the simulation and test category at the 1986 IEEE/ACM Design Automation Conference (DAC), the US National Science Foundation's Research Initiation Award in 1989, and the IEEE/ACM Design Automation Scholarship in 1990 and 1991.