ORCHID: A MALICIOUS WEBSITE DETECTION SYSTEM

EUGENE VAHLIS AND LI YAN

Abstract. With the web becoming the dominant platform for information distribution and retrieval came a new information security threat in the form of online exploits. These are made in attempts to gather private information or gaining control over machines connected to the internet. Orchid is an open source system implemented in python for finding and classifying malicious pages on the web. In this paper we describe orchid's architecture, the algorithms used and several examples of usage for detecting malicious pages.

1. Introduction

If we look at the WebSense security alerts page[1] we can see that new web based exploits are found almost every day. These exploits range from simple social engineering to complex use of bugs in the client's browser software. One example of this is the JS/Wonka technique (which we explore in more detail in section 5) which allows attackers to obfuscate their malicious code on a compromised site thereby reducing the chance of the code being detected.

Our system consists of three components: a crawler (which is described in section 3) , an analyzer (section 4) and rules (section 5). The three components work together in a multi threaded environment to detect and classify web pages that we think may contain malicious code.

A simplistic description of the system's operation would be:

(1) The user feeds the controller a list of URLs which act as a "seed" for the crawler.
(2) The crawler extracts the links from the URLs in the queue and passes them to the analyzer.
(3) The analyzer performs analysis on the pages extracted by the crawler, catalogs the results (i.e. if an exploit was discovered) and adds the links that appeared in the page to the crawler's queue.

As we will show in the following sections, the system is extremely easy to use and extend and is very flexible. For our experiments we used one machine on which we ran the crawler (using 5 threads). The fact that only one machine was used severly limited the portion of the web that we were able to crawl. We believe that if the enhancements described in section 6 are introduced the performance and results of the system will improve significantly.

The work is organized as follows: in section 2 we describe the origin of our idea and a project by Microsoft called "Honey Monkey" with a similar goal but a different approach, in section 3 we describe the design and algorithms of our crawling engine "Orchid", in section 4 we describe how the system processes acquired URLs and how to extend it, in section 5 we give some real-world examples of online exploits and introduce the rules by which our system can identify pages containing them

and in section 6 we describe several possible improvements to our system which we believe can improve it's performance and results.

2. Related Work

The idea for our project was inspired by the Honey Monkey project [2] by Microsoft. Honey Monkey aims to discover malicious websites by running several virtual machines on each of which an instance of the Microsoft Internet Explorer software was running. The virtual machines were controlled in such a way as to simulate a "real person" browsing the web.

The way Honey Monkey discovered malicious sites was by recording every action performed on the system and analyzing the collected data. The browsers are the only active programs running on every virtual machine and so any modification done to the system is done by the browser. When a "serious" modification was detected the system signalled an exploit. This approach allows for detection of zero-day vulnerabilities as well as existing ones without the need for creation of a database of signatures to identify the exploits.

Our approach is different. Our system crawls the web using a lightweight robot and analyses the contents of each page in static mode. The fact that we are doing a static analysis of the web removes the need for a virtual machine to run the system because no execution of remotely supplied code is performed.

While Honey Monkey's goals were to identify exploits that compromise the system, our goal is more extensive. While our system supports detection of known exploits we also believe that web sites that host other, potentially non-malicious, code such as the "pop-under" javascript which is described in section 5 are likely to host malware as well. These sites attempt to manipulate the client and the browser software to view content which the client did not intend to and therefore clearly do not have good intentions.

The only requirement to run our software is a python interpreter. Therefore, it is very easy to scale and can be used, if parallelized, to scan large portions of the web quickly, while the Honey Monkey project would require much more resources and probably dedicated machines because Virtual Machine software has a very high resource consumption.

3. The Crawler

3.1. Introduction And The State Of The Art.

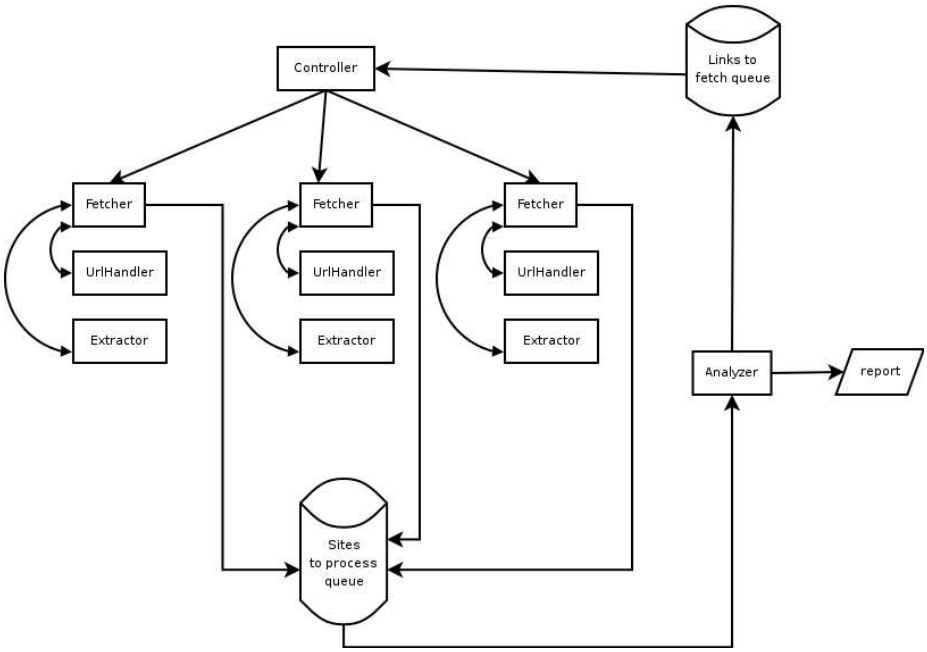
There are many "crawlers" or, as they sometimes referred to, "robots" publicly available on the web, the most popular ones being Nutch [4] and Harvest Man [3]. However, we discovered that most of the crawlers are specialized in some aspect of data mining, poorly documented or both.

When writing the crawler our goal was to create a generic, well documented and easy to use piece of software which can be used to collect any taxonomy on the internet that can be implemented in python. The Orchid crawler is released under the MIT License [5] and can be used in the ways described there.

There are many difficult points in writing a crawler: load balancing, being friendly to innocent web servers, broken HTML and many other technical issues. To overcome all these problems we followed the guides [6] and [7].

One major decision in writing a robot is whether to use a single or multiple threads for fetching page contents. We decided to use multiple threads. Our main

FIGURE 3.1. Orchid Control and Data Flow



reason for choosing multiple threads was that many web servers have high latency and while one thread is waiting for a slow server another free fetcher can fetch several other pages, therefore increasing the crawling speed significantly.

3.2. **Architecture.** The Orchid crawler consists of three programs: controller, fetchers and analyzer, and three databanks: The *linksToFetch* queue, the *sites* queue and the database of analyzed data (we will refer to it as the *db* from now on). The following is a description of the roles of each part and data structure, the control and data flow are visualized in figure 3.1.

Controller. The controller is a single thread which is responsible for controlling the fetchers by selecting a URL to crawl (using the analyzers' *selectNextUrl* method, finding a free fetcher thread and assigning the selected URL to the fetcher. Here is a pseudo-code description of the controllers' action:

Algorithm 3.1. *Controller pseudo-code*

```
while (number of pages crawled < limit):
    url = select the next url to crawl
    fetcher = null
    for f in fetchers:
        if f is free:
            fetcher = f
    assign url to fetcher
    wake fetcher
```

Fetchers. The fecthers are a set of threads responsible for retrieving a URL's contents and extracting the hyperlinks from the contents. The fetcher uses two other components for it's operation: *UrlHandler,* which retrieves the contents of a URL and *OrchidExtractor* which extracts the links and contents of the retrieved page. Each fetcher processes one URL at a time. Every time a URL is assigned to it by the controller the fetcher uses the *UrlHandler* class to download the contents of the page and the uses *OrchidExtractor* to process these contents and create a *Site* object. The *Site* object is inserted into the *sites* queue and the analyzer is awoken. Here is a pseudo-code description of these actions:

Algorithm 3.2. *Fetcher pseudo-code*

```
url is assigned by the controller
contentsStream = urlHandler.processUrl(url)
extractor.setSite(contentsStream)
extractor.extract()
processedInfo is the information generated by the extractor
site = Site(processedInfo)
add site to the site queue
wake the analyzer
```

Analyzer. The analyzer is the main data mining module of the system. The Orchid crawler contains a naive analyzer which only adds links to the *to fetch queue* without processing them at all. To use the Orchid crawler one would have to subclass *NaiveAnalyzer* and implement the following three methods: *analyzeSite, addSiteToFetchQueue* and *selectNextUrl.* All the synchronization is done in the *run* method and therefore knowledge of multi-threaded programming is not necessary to use Orchid.

The roles of the above methods are as follows:

- *analyzeSite*: This action is invoked on a site from the *site* queue. In this method should be all the processing of the site and the separation of the links that should be followed some time in the future from those that should never be followed. The links that should be followed should be stored in some datamember of the analyze instance.
  The naive implementation simply marks the URL of the site as visited and puts all the new links in the list of links to be added to the crawl list.
- *addSiteToFetchQueue*: This action is invoked on the *links to fetch queue* and should be relatively fast[1]. Here should be the code that adds the links that should be followed to the *links to fetch queue.*
  The naive implementation classifies each link by the URL of it's sever and adds them all the the queue, which in this case is a map of *server url* $\rightarrow$ *queue of links.*
- *selectNextUrl*: This action is invoked by the Controller (as opposed to the other two which are invoked by the analyzer's *run* method) and should contain the code for choosing the next URL to crawl. It should always

[1]The reason for having two separate methods is for future extensions. If some analyzer implementation will require a long time to process a site there is no need to lock the queue and prevent the controller from assigning URLs to fetchers.

return one string URL.
The naive implementation chooses a random server and then takes a URL from it's associated queue. This approach is described in more detail in section 4.

The main method of the analyzer performs the following actions: waits until a site is inserted into the *site* queue, calls *analyzeSite* to perform the necessary data mining logic, calls *addSiteToFetchQueue* to update the *to fetch queue* and loops. Here is a pseudo-code description of the operation of the analyzer's *run* method:

Algorithm 3.3. *Analyzer pseudo-code*

```
while (there are more sites to crawl):
    fetch site from site queue
    call analyzeSite
    call addSiteToFetchQueue
```

Our extended security-based content analyzer (called *malcontent*) is described in greater detail in the following section.

4. The Malicious Content Analyzer

4.1. **General description.** The section deals with the way our security based content analyzer is designed and how to use and extend it. The goal of our analyzer is to identify and classify web pages containing *malicious code*. The definition of *malicious code* may vary and is defined completely by the set of *Rules* specified to the analyzer at it's construction.

The information available to the analyzer during runtime is, for every web page, it's URL, links and content (if of mime type either *text/html* or *application/javascript*), therefore, anything that can be inferred using python code and this information can be inferred during the analysis step of the analyzer's main loop.

The classification and identification of malicious content is done by applying *rules* to the currently processed page. The rules are discussed in more detail in the following item. In general, each rule identifies a certain class of vulnerabilities, for example: a rule which matches a regular expression against the pointer of a certain type of links.

4.2. **What are rules?** Rules are the core of the Orchid malicious content detection system. The rules are divided into classes that can be analyzed by a generic piece of code. The malicious content analyzer applies all the rules to every page it encounters. Each rule analyzes the site in it's own way and determines whether it contains malicious content. If the page is determined to be malicious the information is stored and various statistics are updated. We have defined an abstract class called *Rule* from which all the concrete rule classes should inherit.

The abstract rule class defines three levels of "maliciousness" of pages: Good, Maybe Evil and Evil. Each site is initially assigned the tag Good. Every time a rule is applied to the site and matches, the rule's associated level is checked against the current level of the site and if it is more severe the site's level is updated to the rule's level. This is described by the following pseudo-code:

Algorithm 4.1. *Malcontent application of rules to sites*

```
s is the site currently being processed
for rule in rules:
    if rule(s) > s.level:
        update the site's maliciousness level
```

The three defined levels can be roughly described as follows: GOOD refers to pages which do not contain any malicious code that we care about. MAYBE EVIL refers to pages which contain code which, under most circumstances, will result in an undesired behaviour of the browser or may signify the presence of a serious exploit. EVIL refers to pages which most definitely are malicious and contain a specific exploit which we can identify.

In the following items we will describe several rule classes we created for our experiments and how one would approach writing new classes of rules.

4.3. **Existing classes of rules and how to write new ones.** In order to be able to perform the experiments described in section 5 we defined three classes of rules: Link rules, Content rules and External IFRAME rules. The rule creation aspects of these rules is described below and the exploits they are targeted at are described in more detail in section5.

Each rule class implements the python meta-method __call__(self, site) where all the processing occurs. Implementing the __call__ meta method allows for the rules being applied to sites (rule(site)). This is an implementation of the COMMAND design pattern.

The type of processing done in the __call__ method is entirely up the the developer. For example: in the LinksRule class the constructor is supplied with a map $re \rightarrow$ (exploit name, link types, level). The regular expression is compiled and the modified map is stored as a data member. During the execution of the rule we iterate over the link types supported by this instance of LinksRule, try to match their URL to the regular expression and if a match is found we update the level of the site to the maximum between the site's level and the rule's level.

In the ContentRule class the definitions of exploits are much more general and therefore more powerful but also more difficult to define in a correct way. In this type of rules a regular expression map is supplied in the constructor, like in the LinksRule class. However, during execution, the regular expression is matched against the raw text of the web page rather then against specific elements.

The ExternalIframeRule class is an example of a case where the sought exploit cannot be identified by merely matching a regular expression against a part of the page. In this case we are trying to detect IFRAME elements in the page which load content from another domain (the reasons for this are described in more detail in section 5). This class of rules identifies only one exploit (as opposed to the other two which can be used to identify a very broad set of exploits), but it demonstrates that Orchid can be used to detect malicious content which requires a more powerful logic than just regular expressions.

4.4. **Crawling strategy.** The decision to which URL to crawl next is highly important. If we simply crawled in a FIFO manner we would send many request sequncially to the same server. The server would detect this, suspect an attempted denial of service attack and block us. One possible way to reduce the load on individual servers is to choose a random domain from the URL queue and crawl one of it's pages. This approach results in a very low frequency of requests to every server

and, in addition, in a very fast growth in the number of servers we have to choose from. The Malcontent analyzer uses this approach.

Another approach that we tried was selecting a random server but giving higher priority to servers on which we previously found a malicious page. This approach did not perform well during our experiment. We believe that the reason for the poor performance was that most bad pages will contain malicious code on their "primary" page (the page that everyone links to) because this page will attract most of the visitors.

5. Examples Of Rules And Exploits

In this section we will present three web exploits of different classes. We will discuss with some detail the vulnerabilities targeted by each exploit. We will also describe our way of identifying these exploits and the rules we created to do so. Finally, we will present the experiments we performed and the results we got.

5.1. The JS/Wonka obfuscation technique.

5.1.1. *The exploit.* JS/Wonka is a fairly new technique which becamse very popular during October 2005. This is not an exploit in itself but rather a way of obfuscating malicious code so that it will be more difficult to identify.

The obfuscation is done using the javascript *escape* and *unescape* functions. The malicious code is encoded using *escape,* then, the encoding along with a call to *unescape* is placed on the compromised website. Here is an example:
Suppose that the code we want to place on the page without being detected is:

```
<iframe src="http://www.XXXXXX.com" width=0 border=0 height=0></iframe>
```

then, we encode it using *escape*:

```
%3C%69%66%72%61%6D%65%20%73%72%63%3D%22%68%74%74%70%3A%2F%2F%77%77%77%2E%58%
58%58%58%58%58%58%2E%63%6F%6D%22%20%77%69%64%74%68%3D%30%20%62%6F%72%64%65%7
2%3D%30%20%68%65%69%67%68%74%3D%30%3E%3C%2F%69%66%72%61%6D%65%3E
```

and place the following javascript on the compromised website:

```
<Script Language='Javascript'>
<!--
document.write(unescape('%3C%69%66%72%61%6D%65%20%73%72%63%3D%22%68%74%74%70
%3A%2F%2F%77%77%77%2E%58%58%58%58%58%58%58%2E%63%6F%6D%22%20%77%69%64%74%68%
3D%30%20%62%6F%72%64%65%72%3D%30%20%68%65%69%67%68%74%3D%30%3E%3C%2F%69%66%7
2%61%6D%65%3E'));
//-->
</Script>
```

Now, if someone opens our site an IFRAME of size 0x0 will be created in which a malicious page is loaded. More details about JS/Wonka are available in [8].

5.1.2. *The detection rule.* To detect this exploit we used the ContentRule class. We classified every page which matched the regular expression

```
unescape\s*\(\s*'[^']*\%3C\%69\%66\%72\%61\%6D\%65[^']*'\s*\)
```

This regular expression would match any page which tries to unescape some sequence containing an IFRAME element. Although it is possible that someone will try to do that without malicious intent, we think it is highly unlikely.

5.2. **Cross Site Scripting.**

5.2.1. *The exploit.* Cross site scripting is a well known web exploit. It is based on the fact that many web pages display parameters they receive in requests without "sanitizing" them. For example if we have a page that receives a parameter *name* and displays "hello *name*" then if a <script> tag is passed as the name the script will loaded and executed by the browser. Cross site scripting (or XSS) is discussed in much more detail in [9] and [10]. One way to use XSS is to place a link on a malicious web site, which will direct the use to a vulnerable page with a malicious script as one of the parameters, for example:

```
<a href="http://www.vulnerable.com/?name=<script>send cookies to
attacker</script>">A very nice site</a>
```

5.2.2. *The detection rule.* We used the LinksRule class and the regular expressions provided at [10] to detect the type of XSS which appears on malicious pages. In order to find XSS attempts we scanned A, IMG and IFRAME elements for pointers matching one of the following regular expressions:

```
((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)
```

or

```
((\%3C)|<)((\%69)|i|(\%49))((\%6D)|m|(\%4D))((\%67)|g|(\%47))[^\n]+((\%3E)|>)
```

The first expression matches <script> tags and the second expression matches <img> tags which contain javascript in their src attribute.

5.3. **External IFRAME spyware and adware.**

5.3.1. *The exploit.* IFRAME is an HTML element which can be placed inside a page and have external content loaded into it, these elements are usually used for advertisement. By External IFRAME we refer to IFRAME elements which load data from a domain different from the domain of the current page. The reason we chose to identify external IFRAMEs is because they are very often used to collect information on an internet user's browsing habits (i.e. spyware) and are also used by most exploits (like the JS/Wonka example we gave above).

When a browser sends a request to a web server the server can record the IP address of the browser. If some server owner (like an advertiser) manages to place IFRAME advertisements in many pages around the web he can collect information about which IP addresses visit which pages.

Another malicious use of IFRAMEs is when an attacker manages to compromise a web page but does not wish to insert big amounts of code to the page to avoid detection. The attacker can instead create a single 0x0 sized IFRAME element pointing to his own page which contains all the malicious code he needs. In this case it is almost certain that the IFRAME pointer will point to some other domain on which the attacker hosted his malicious page.

5.3.2. *The detection rule.* To detect external IFRAMEs we wrote a separate rule class ExternalIframeRule. When applied to a web page this rule iterates over all the IFRAME elements and compares the domain of the pointer to the domain of the page currently being analyzed. If the domains are different the rule matches.

This rule is quite simple but it demonstrates Orchid's support for detection of exploits which cannot be discovered by simply matching regular expressions.

5.4. **Pop under windows.**

5.4.1. *The exploit.* "Pop under" windows is a technique commonly used by "seedy" websites to open pop up windows without the user noticing them. The technique can be used to display advertisements as well as to exlpoit browser vulnerabilities. Pop under windows are windows which appear beneath the current broswer window. Here is an example of opening a pop under window:

```
window.open('http://www.seedysite.com');window.focus()
```

This opens seedysite.com in a new window and returns the focus to the current window.

5.4.2. *The detection rule.* For detecting pop under windows we used the ContentRule class. We scanned every page for the following regular expression:

```
window\s*\.\s*open.+window\s*\.\s*focus
```

This will detect sequences of opening a new browser window and returning the focus to the original window.

5.5. **Experiments.** To demonstrate the abilities of our system we performed several experiments of malicious website detection.

5.5.1. *Set up.* In our experiment we used all the rules described above to scan 14000 web pages for malicious code. Our "seed" (initial URL list) was several software piracy and pornography sites which we found on Google.

Before running the experiments we tested Orchid on sites which we know to contain each of the exploits described above. For example: www.free-daily-jigsaw-puzzles.com is known to contain the JS/Wonka exlpoit (Warning: do not try to load this site as your system may be compromised as a result). Orchid was succesful in identifying everyone of the exploits during the testing phase.

During the experiments themselves we discovered that a very large portion of the sites that we scanned contained external IFRAMES and pop under windows. This is to be expected because these techniques do not necessarily harm the user but simply annoy him or covertly collect data on him.

To run the experiments we used the following parameters:

- Maximal number of pages: 14000
- Number of fetcher threads: 5
- Socket timeout: 15 seconds
- Delay between URLs being assigned to fetchers: 2 seconds

5.5.2. *Difficulties we encountered.* Our initial version of the crawler didn't select a random domain to crawl to but rather crawled on a first-in-first-out basis. This approach was not succesful because the frequency of requests to each server was too high. This problem was resolved by the approachs described in section 4.

Another problem which we encountered was that, altough we reduced the request frequency to each server, the requests to the DNS server were still coming too fast which caused the DNS server to block us. This was resolved by reducing the crawling speed.

5.5.3. *Results.* Here are the results that we got after crawling 14000 pages (log files are available in the package):

- Number of pages succesfully scanned: 12659
- Maliciousness level breakdown:

	Total count	Portion of scanned pages
Good	11782	0.931
Maybe Evil	853	0.067
Evil	24	0.002

- Specific rule match count:

	Total count	Portion of scanned pages
External IFRAME	2092	0.165
Pop under window	151	0.012
XSS	29	0.002
XSS with IMG	3	0.00024
JS/Wonka	2	0.00016

- Time required to perform scans: Approximately 20 hours.

5.5.4. *Analysis of results.* Te results we got were what we expected. As previously mentioned, we used a number of pornography sites as our initial seed. As expected, those sites contained many external IFRAME elements (mostly advertising other sites of similar content).

The second most prevalent rule match was for the pop under windows. This is also expected as this technique is very easy to use and does not require knowledge of browser vulnerabilities. Most of the pop under windows are simply advertisements meant to force the user to continue using a certain site, but quite a few of them attempt to install software on the user's machine using methods such as ActiveX controls which are enabled in old versions of Internet Explorer.

The number of XSS matches was higher than we expected. We believe this is partially due to some false positives.

An interesting results of the experiment was that the vast majority of the crawled pages contained adult content. This is due to the fact that websites of such nature tend to contain a large amounts of links to other such sites and have a very strong lack of diversity in content.

Unfortunately, we were not able to discover any new sites containing the JS/Wonka exploit (the two matches above are for sites that we found and inserted into the seed list. We believe that if the number of pages to crawl was increased and if websites of other content types were included in the seed list the results would be better.

6. Possible Extensions

6.1. **Parallelization.** One of the factors that most limits Orchid's crawling rate is the number of requests it sends per minute. At it's current state Orchid runs on a single machine (with a single IP) and therefore the maximal crawling speed is quite slow. If we tried to crawl faster servers would block our requests to reduce their load.

One good way to solve this problem and siginificantly increase the crawling speed of Orchid is to parallelize it's operation. Writing parallel algorithms is quite difficult and there are many resources on the web on how to do it correctly ([11] and [12] are good examples). If parallelized, Orchid could be used for large scale web analysis.

6.2. **Improved detection heuristics.** The current set of rule classes in Orchid supports only direct analysis of a web page. We believe (without justification) that there are some web page parameters which have non-zero correlation with the page's "maliciousness". For example: it is not unreasonable to assume that a certain page's backlinks (links that link to that page) and it's contents can indicate whether it contains a certain exploit.

If such a correlation indeed exists a machine learning model can be trained on the data collected by the direct rules and then used to identify new exploits. One model which may be suitable for this task is the Support Vector Machine[13, 14] model.

7. The Orchid Software Package

The Orchid system comes as tarred and gzipped file called "orchid.tgz". To use the package first go to some directory and extract it:

```
cd my_useless_stuff
tar xvzf orchid.tgz
```

Now, there are several important files: "orchid/orchid.py", "orchid/malcontent.py" and "orchid/documentation/index.html". The documentation is very pleasant and well formatted HTML which resembles Javadoc. It was generated by the excellent epydoc[15] package.

To run our experiment you should do:

```
cd orchid
python experiment.py
```

Please note that our initial seed list may contain URLs with offensive words due to the nature of such sites.

8. Conclusion

Browsers have become one of the most important category of programs for computer users and therefore it is highly important to make them secure. We hope that our system can demonstrate one way of finding what to fix or improve in browsers and what to be careful of.

References

[1] WebSense security alerts `http://www.websensesecuritylabs.com/alerts/`
[2] Honey Monkey, Microsoft `ftp://ftp.research.microsoft.com/pub/tr/TR-2005-72.pdf`
[3] HarvestMan Web Spider `http://harvestman.freezope.org/`
[4] Nutch Web Spider `http://lucene.apache.org/nutch/index.html`

[5] MIT License http://www.opensource.org/licenses/mit-license.php
[6] Sriram Krishnan's guide to writing crawlers http://dotnetjunkies.com/WebLog/sriram/archive/2004/10/10/28253.aspx
[7] SearchTools crawler checklist http://www.searchtools.com/robots/robot-checklist.html
[8] WebSense security analysis of JS/Wonka http://www.websensesecuritylabs.com/resource/pdf/wslabs_wonka_analysis_oct05.pdf
[9] Cross site scripting explained, Amit Klein http://crypto.stanford.edu/cs155/CSS.pdf
[10] Detection of SQL Injection and Cross-site Scripting Attacks http://www.securityfocus.com/infocus/1768
[11] Parallel Algorithm Design http://www.dcs.ed.ac.uk/home/stg/pub/P/par_alg.html
[12] Designing Parallel Algorithms http://www-unix.mcs.anl.gov/dbpp/text/node14.html
[13] Support Vector Machine, Wikipedia http://en.wikipedia.org/wiki/Support_Vector_Machine
[14] Learning to Classify Text using Support Vector Machines, Thorsten Joachims http://www.cs.cornell.edu/People/tj/svmtcatbook/
[15] Epydoc http://epydoc.sourceforge.net

Dept. Computer Science, University of Toronto, 6 King's College Rd., Toronto, Ontario, M5S 3G4, CANADA
E-mail address: evahlis@cs.toronto.edu

Dept. Computer Science, University of Toronto, 6 King's College Rd., Toronto, Ontario, M5S 3G4, CANADA
E-mail address: liyan@cs.toronto.edu