Web Spam Detection for Heritrix

Ping Wang

May - August 2008

This project is one of the four projects mentored by Internet Archive in Google Summer of Code 2008.

1 Introduction

Web spam is a big problem for today's search engine. Some spam websites do not contain any useful information. Crawling those websites is just a waste of time, effort and storage space. Heritrix is not primarily a search engine crawler, it is alright for it to have a little representative spam. However currently web spam takes up too much of the resources, proportionately, therefore it is necessary for Heritrix to be able to detect web spam during crawling.

Today all kinds of techniques have been employed by spam websites in order to get higher rankings than they deserve and hide their identities. These techniques can be categorized as boosting or hiding [1]. While content and link spam are most widely used boosting techniques, and redirection and cloaking are hiding techniques. At the same time, many web spam detection techniques have been proposed in literature. Some of them use features such as hyperlink structure between pages and the DNS records of hosts, and some content-based methods need training stage before detection. They do not fit well for web spam detection for Heritrix, because Heritrix is not a search engine, doing complex analysis on the fly, such as analyzing hyperlink structure and web page content would highly increase the overhead, and affect its main functionality - website crawling. We propose to focus on detecting JavaScript redirection- and cloaking-based spam for two reasons: (1). These two spam techniques are prevalent currently. (2). The detection approaches discussed in the next section are simple and effective, and cause less overhead.

Furthermore, there is one other reason that this project is of interest to the Heritrix community. As we will discuss in methodology section , in order to detect JavaScript redirection spam and cloaking-based spam, JavaScript-execution capability is required, and this capability will also help discover legitimate outlinks of a web page.

2 Methodology

In this section, we briefly explain JavaScript redirection and cloaking spam techniques along with corresponding detection approaches. These detection approaches discussed here were proposed in [1, 3]. For more information, please refer to those papers.

2.1 JavaScript Redirection

Spammers usually build numbers of doorway web pages, which will redirect to the real spam website. Redirection can be achieved by using HTTP status codes, META refresh and JavaScript. JavaScript redirection is widely used in web spam and hard to detect by static analysis. The following is an simple algorithm to detect JavaScript redirection, if a browser is used.

```
Load the web page twice with JavaScript enabled and disabled;
if DstURL(E) = OrgURL, then
  No redirection;
else if DstURL(E) != DstURL(D) then
  JavaScript redirection;
else
  No JavaScript redirection;
```

where OrgURL, DstURL(E) and DstURL(D) represent original URL, destination URL with JavaScript enabled and destination URL with JavaScript disabled respectively.

However, we don't need to load the page twice, because we can tell if JavaScript redirection exists by evaluating JavaScript code within the page during parsing. Therefore what we need to do is:

```
Step 1: Load the page;
Step 2: Parse the page with JavaScript enabled;
Step 3: if JavaScript Redirection exists
            mark the page, and take corresponding actions;
        else
            continue the normal process;
```

In Step 3, after detection of JavaScript redirection OrgURL=>DstURL(E), some simple rules can be applied to rule out common non-spam redirections, for example:
- www2007.com => www.2007.com/;
- www2007.com => www.2007.com/index, default, ...htm, html, asp, aspx, php, ...;
- Redirection within the same host.
The rest of redirections are considered as JavaScript redirection spam.

Of course there will be false positives. To be more accurate, manual investigation or other web spam detections can be applied offline.

2.2 Cloaking

The basic idea of cloaking is that serving different content to different visitors. Web server can decide which web page to serve based on User Agent field, IP addresses of the visitor and etc. Web pages can contain scripts which will rewrite the content, such that users will see rewritten content by using a web browser, but a crawler may not be able to see the same thing, because usually most crawlers do not execute scripts.

Click-through cloaking is another cloaking techniques. It can be either on the server side or the client side. A server-side check of Referer field in the HTTP header is done if the spammer own the website that host the spam URL; if the spam URL locates on a free hosting website, a client-side check of the browser document.referrer object is performed, and then based on the result different pages are served by the website hosting server in the first case or displayed by using document.write() in the second case.

Therefore we can detect click-through cloaking-based spam website in the following way, which also requires the crawler be able to interpret scripts:
Setp 1. First visit - access the website in the normal search engine way, and obtain the content of the page content(engine);
Setp 2. Second visit - make the access appear to come from a search click-through for both server-side and client-side referer checks, and obtain the content of the page content(click-through);
Setp 3. Compare the content difference between two visits diff(content(engine), content(click-through)). If there is a big difference, then this website is highly suspicious to be spam website, and needs further investigation, otherwise label it as benign.

In this project, we focuse on detecting click-through cloaking.

3 Third Party Libraries

3.1 Embedded JavaScript Engine

As it is discussed in methodology section, both detection approaches for cloaking-based and JavaScript redirection-based web spam require a web crawler be able to execute inlined/external JavaScript code.

Rhino [4] is an open-source JavaScript engine written in Java under the MPL 1.1/GPL 2.0. We will embed it in Heritrix.

3.2 HTML Parser

To run JavaScript code in a web page, an embedded JavaScript engine is not enough, we need to get all the objects and attributes associated with them of a web page. In other words, we need a HTML parser which is able to parse HTML documents and generate the HTML Document Object Model (HTML DOM).

Cobra [5] is a HTML render and parser written purely in Java. Its parser is able to generate HTML DOM tress. Cobra embeds Rhino JavaScript Engine, such that during parsing, inlined and extenal JavaScript code can be run, and the corresponding modificaiton on DOM made by those code will be reflected in the resulting DOM.

In this project, we choose to use Cobra as the HTML parser, and also will utilize its ability of running JavaScript code with the help of Rhino.

4 Designs and Implementation Details

4.1 JavaScript(JS) Execution

An integration challenge was brought by Mr. Mohr: how to be compatible with existing work flow of Heritrix?

> "How to ensure that all requisite resources are fetched (included SCRIPTs, and possible CSS) before Javascript is executed, but in a way roughly compatible with the existing discovery/flow of URIs in the crawler. (For example, so that every URI fetched is written verbatim in the crawl's W/ARC files. Or, that clear rules apply when there are conflicts between whether a URI is needed for JS execution, but ruled out by the crawl operator's other scope/settings.)"
> "Does downloading these additional resources go through the normal Heritrix scheduling, politeness, analysis, etc. steps? On the one hand, we want the crawler's logs and (W)ARC files to be an accurate record of the resources it actually retrieved. OTOH, we don't want to have the resources we need for JS/DOM processing on a single page be scheduled far in the future, or their loading aborted by other crawler settings."

Because JS code could be internal (inlined in the HTML document) or external (i.e. <script src="location" >). In cobra HTML parser, whenever an external JS code is needed, it will try to fetch the source code within the parser. We would like to have every fetch follow the processor chain, i.e go through normal Heritrix scheduling, politeness, analysis, etc. steps, and this implies we need to fetch required external JS resources before JS execution. Moreover, we don't want to avoid double fetching for an HTML document, so during JS execution, we need to be able to retrieve contents of the HTML document, and required JS resources as well. In order to do this, we introduce a caching mechanism - fetch cache, which is used to preserve these kind of information.

. Class FetchCache

The fetch cache contains three big maps: resource map, dependent map and content or content location map. In the following, resource means requried JS resource, i.e. JS source file, and its link is extracted from an HTML document.

Resource map: <uri of a resource, the list of HTML documents which depend on this resource>;
Dependent map: <uri of an HTML document, the list of resources on which this HTML document depends>;
Content or content location map: <uri of a resource or an HTML document, content or content location of this uri>.

Actually in my implementation, I split Dependent map into two maps:
Dependent map1: <uri of an HTML document, the list of unfetched resources on which this HTML document depends>;
Dependent map2: <uri of an HTML document, the list of fetched resources on which this HTML document depends>;

There is an additional map - waitToSchedule, used to save a set of HTML uris whose resource are all available, and the uris of its resources. These HTML documents are ready for parsing and JS evaluation.

. Class FetchCacheUpdater

FetchCacheUpdater is a post-processor, similar to CrawlStateUpdater, but it is used to update fetch cache after a uri has been processed by LinksScoper. The reason FetchCacheUpdater has to be performed after LinksScoper is that, some out links of a document may be out of scope, and there may be some JS required resources; LinksScoper will make those out-of-scope links, then in FetchCacheUpdater, we can easily find those required but out-of-scope resources, and log these information for user.

Another job of FetchCacheUpdater is creating special crawl uri which is designed for JS execution purpose. After updating fetch cache, FetchCacheUpdater will try to get all HTML urs which are ready for JS evaluation, and create a crawl uri whose scheme is "x-jseval" for each of them, (i.e, if a normal uri is http://www.uri.com, then the special uri would be x-jseval:http://www.uri.com), and then add them to outCandidates of current uri for later scheduling.

. Class ExecuteJS

This processor is used to parse an HTML document with JS enabled, and also simulate HTML events in the document, and try to discover new links created by JS code. It only accepts uris with scheme of "x-jseval".

The content of the HTML uri being processed and its JS required resources can be retrieved from fetch cache.

Regarding HTML events, onload is simulated during parsing, onclick, onmouseover and onmouseout are handled after parsing. These three types of HTML events can be simulated in an accumulative fashion, or from a fresh DOM (created after parsing) everytime.

Discovering new links is performed after parsing and after each HTML event simulation. New links will be added to outLinks of the HTML uri.

4.2 JavaScript Redirection Detection

Detecting JS redirection is easy, we only need to check if the document location in resulting HTML DOM after parsing or simulating HTML events is different from the original location (the location we fetched the document).

. Class DetectJSRedirection

This functionality is implemented in processor DetectJSRedirection.

If processor ExecuteJS is included in the job profile and performed before DetectJSRedirection (usually this is the case), we only need to check if there is a new link whose context is “JSRedirection” in outCandidates of current uri, if so JS redirection happened, otherwise no JS redirection, because in processor ExecuteJS, context of JS redirection link is set to “JSRedirection”.

Processor DetectJSRedirection can also work without processor ExecuteJS. In this case, JS code needs to be evaluated in order to detect JS redirection, so all JS required resources must be available before detection.

No matter which is the case, DetectJSRedirection can only accepts crawl uri with scheme of “x-jseval”.

4.3 Cloaking Detection

. Class DetectCloaking

Cloaking detection is implemented in processor DetectCloaking, and this processor currently only focuses on click-through cloaking. As mentioned in methodology section, to detect client side click-through cloaking, we need to compare resulting documents obtained after executing JS code with document.referrer set to empty with document.referrer set to something; if they are different, we decide that client side cloaking happened. To detect server side click-through cloaking, we need to compare documents fetched with referrer field in HTTP header set to null or set to something; if they are different, server side cloaking happened.

Therefore, some information must be available, in order to detect click-through cloaking:
For client side cloaking detection
- referrer
- resources: The list of resources required by JavaScript execution.

For server side cloaking detection
- content-with-referrer: fetched content with referrer field set to non null in HTTP header.
- content-without-referrer: fetched content with referrer field set to null in HTTP header.

As we can see in order to detect server side click-through cloaking, we need to fetch the same page twice with different HTTP header settings, of course we would like to have these two fetches go through Heritrix processor chain; and we also need to preserve those required information mentioned above. So to address this problem, we use the similar way as we did for JS execution, we introduce another cache - cloaking cache, which has a map, holding all required information.

. Class Cache4CloakingDetection

Map: <uri of a HTML document, requried prerequisites>

Requried prerequisites: “referrer”, “js-required-resources”, “content-with-referrer”, and “content-without-referrer”.

The task of updating cloaking cache is performed within DetectCloaking.

5 Major Algorithms

5.1 JS Execution

. Processor chains

Preprocesses − > fetcher − > extractors − > ExecuteJS − > ARCWriter − > CrawlStateUpdater − > LinksScoper − > FetchCacheUpdater − > FrontierScheduler

. Algorithm implemented in FetchCacheUpdater

For detailed explanations of maps in FetchCache, please see FetchCache in designs and Implementation details section.

curi is current uri being processed.

```
if curi is not fetched successfully and it is a script resource,
then remove all information related to it from fetch cache;
else if curi is successfully fetched, then {

    if curi is a script resource, then {

        if curi is in resource map then {
            for each HTML uri depending on it {
                find the HTML uri in dependent map;
                update its resource list, i.e. remove curi from its list;

                find the HTML uri in candidate map;
                update its resource list, i.e. add curi to its list;

                if all resources are availabe {
                    add curi to the waitToSchedule list;
                    remove corresponding entry in both dependent
                    and candidate map;
                }
            }
            remove the entry of curi in resource map;
        }
        add an entry in avaliable map (curi, content/content location);
   }

  if curi is a HTML uri then {
      create a collection of required resouece uris from outCandidate collection;
      for each resource uri in the collection of JS required resource uris {
          if it is in available map then {
              remove it from the collection
          }
          else {
              if it is in resource map then add curi to its dependents list;
              else add an entry in map1 (resource uri, curi);
          }
      }
            if the collection required resource uris is empty then
          add curi to the waitToSchedule list;
      else {
          add an entry in dependent map (curi, collection of required resource uris);
          add an entry in cadidate map;
      }
      add an entry in avaliable map (curi, content/content location);
  }
}
At last create special crawl uri for each member in waitToSchedule list,
and put them in the outCandidates of curi;
```

5.2 JS Redirection Detection

. Processor chains

Preprocesses − > fetcher − > extractors − > ARCWriter − > CrawlStateUpdater − > DetectJSRedirection − > LinksScoper − > FetchCacheUpdater − > FrontierScheduler

. Algorithm implemented in DetectJSRedirection

```
if (ExecuteJS module is added in the profile) then {
    check if JS redirection happened;
    if yes, return true for JS redirection;
    else return false;
} else {
   // Similar to the procedure in ExecuteJS,
   // but only to check JS redirection, not find new links

    DOM dom = parse(HTML document referred by x-jseval crawl uri);
    check if JS redirection happened;
    if yes, return true for JS redirection
    if no, then simulate HTML event one by one from a fresh DOM:

    for every HTML event (onmouseover, onmouseout or onclick) do {
         get a fresh DOM (parse the document again);
         simulate the event;
         check if JS redirection happened;
         if yes, return true for JS redirection;
         else continue;
    }

    return false for JS redirection;
}
```

5.3 Click-through Cloaking Detection

For the meaning of those prerequisites, please see DetectCloaking in designs and implementation details section.

. Processor chains

Preprocesses − > fetcher − > extractors − > ARCWriter − > CrawlStateUpdater − > DetectCloaking − > LinksScoper − > FetchCacheUpdater − > FrontierScheduler

. Algorithm implemented in DetectCloaking

```
if uri is an HTML document then {
    if (context of uri is not null) {
        update Cache4CloakingDetection with (uri, data)
        and data.put("content-with-referrer", content);
        update Cache4CloakingDetection with (uri, data)
        and data.put("referrer", context);
        if (! data.containKey("content-without-referrer") {
            create a new crawl uri with context set to null,
            and force it to be crawled;
        }
   } esle {
       update Cache4CloakingDetection with (uri, data)
       and data.put("content-without-referrer", content);
       if (! data.containKey("content-with-referrer") {
           create a new crawl uri with context set to itself,
           and force it to be crawled;
       }
   }
}
```

```
if uri is an x-jseval uri {
    get required resources;
    update Cache4CloakingDetection with (uri, data)
    and data.put("resources", content);
}

check if all fields (referrer,resources, content-with-referrer
and content-without-referrer) are available;
if yes {
    withReferrerDom = parse(uri document with null referrer);
    withoutReferrerDom = parse(uri document with referrer (itself));
    if (withoutReferrerDom == withReferrerDom) {
        no client side cloaking;
    } else {client side cloaking}

   withoutReferrer = get value of "content-without-referrer"
   from Cache4CloakingDetection for current uri;
   withReferrer = get value of "content-with-referrer"
   from Cache4CloakingDetection for current uri;

   if (withoutReferrer == withReferrer) {
       no server side cloaking;
   } else {server side cloaking}

} else {
   return;
}
```

6 Modifications On Cobra Library

6.1 HTMLParser

In Cobra, the class which performs parsing is HtmlParser. As we mentioned above, when the parser sees a <script> tag, it will try to evaluate the script code. If the script is inlined, the parser can easily read the code from the document input stream, but if the script is an external file, it will try to fetch the script file first, and then do the evaulation, in other words, the fetching is done during parsing. In this project, we introduced fetch cache mechanism to guarantee that all external scripts are fetched before parsing, so we modified the parser to get the content of a external script file locally (already fetched) instead of initiating a new fetching out of Heritrix process chain.

The main modifications are:

1. The class is renamed as HTMLParser instead of HtmlParser, and put it in package org.archive.modules.extractor.jsexecutor instead of org.lobobrowser.html.parser.

2. A new constructor is added. It takes one more parameter than the one we used originally, and the parameter is a HashMap in which contents of external script files are stored, and the keys are the URLs of the scripts.

3. During parsing, when a <script> tag associated with an external source file is seen, the parser will try to find the content of the file by calling member function fillInSourceCode instead of fetching the script.

4. Since in HTMLParser, class ElementInfo, LocatorImpl and StopException in package org.lobobrowser.html.parser are needed, but they were original protected classes, we changed them to public class in order to be able to use them in HTMLParser.

6.2 Archor Tag

The following code is allowed in HTML document, where, the value of attribute href is JavaScript code.

```
<body>
  <script>
  var rootUri = 'cas'+'e2-success';
  function navigate() {
    document.location.href=rootUri;
  }
  </script>
  <a href="javascript:document.location.href=rootUri">href</a><br/>
</body>
```

In Cobra, they only consider the common case that the value of href is a URL. So in order have Cobra HTML parser support the above case, modifications have been made within class HtmlController and HTMLAbstractUIElement.

6.3 JavaScript Functionality

JS Redirection

In Cobra, when a JS redirection happens, the member function setHref in class Location will be called. If a HTML renderer is associated with current application, a new HTML document will be created, fetched and get parsed. In this project, we do not need to render the HTML document, we only need to tell if JS redirection happens or not, so minor modifications have been made in class Location.

setTimeout And setInterval In JS

setTimeout and setInterval, two JS functions are implemented in Cobra by calling member function setTimeout and setInterval in class Window. In this project, we only need to simulate what is going to happen when an HTML event is triggered, accurate timing is not necessary, so these member functions have been modified for this purpose.

7 Discussion and Future Work

7.1 JS Execution

. Content Location in FetchCache

Currenly, in fetch cache, what is saved in available map it the content of each document, and they are all in the memory. Obviously, if a big crawl job is running, sooner or later it will be out of memory, if we keep adding new contents in the fetch cache. To address this problem, a aging mechanism can be designed, i.e. each available document in fetch cache has a age property. An age threshold is predetermined, and when a document is available, not only content but also content location (e.g. arc file name and offset within the arc file) will be stored in the available map, as time goes on, the age of the document increases, once the document's age is above the threshold, the content of this document is cleared out from the memory, only its location is presented. When the document is needed again, its content can be retrieved from the specified location.

. Use JavaScript to import another JavaScript

In ExecuteJS, an HTML document will only be parsed once with JS enabled, and generate corresponding DOM, and do not process again, so in the following case, we will not be able to obtain the final correct DOM.

test.html

```
<html>
    <head>
        <script id="jsfile1" src="code1.js"></script>
        <script id="jsfile2"></script>
    </head>
    <body>
        <P>Header</P>

        <div id=DataContainer>
            <script type="text/JavaScript">
                DoRoutine('code2');
            </script>
        </div>

        <P>Footer</P>
    </body>
</html>
```

code1.js

```
function DoRoutine(routineID) {
    document.getElementById('jsfile2').src = routineID + ".js";
}
```

code2.js

```
function getDATA () {
    var data = new Array(3);
    data [1] = "This is DATA 1, ";
    data [2] = "This is DATA 2, ";
    data [3] = "This is DATA 3";

    document.getElementById("DataContainer").innerHTML =
    data[1] + data[2] + data[3];
}
getDATA ();
```

So to make ExecuteJS behavior much more like a web browser, repeated JS evaluation is necessary. But there are problems need to be solved:
1. How to schedule and fetch new resources appearing the resulting DOM?
2. Even we are able to fetch new resources, how we can preserve current parsing status and continue the JS evaluation, when the new resources are available?

. Simulate HTML events in an accumulative fashion or from a fresh, just-loaded state (a fresh DOM created after parsing)
In this project both ways have been implemented (in the code, the former has been commented out). Simulating HTML events in an accumulative fashion is more efficient that the other way, because for the latter, the original document is parsed every time before simulating a event (There might be better way to implement this). However, since we cannot predict user's behavior, this former way might not be very accurate. I do not have a good idea of how to compare which way is better.

7.2 Cloaking Detection

. Cloaking Techniques
There are a lot of cloaking techniques:

- Click-through cloaking: client side cloaking and server side cloaking;
- Cloaking based on User Agent field or know crawler IP addresses;
- Rewrite web page using scripts, since crawlers usually do not execute scripts;
- ... (I are not aware of)
- Mix these techniques.

In this project, we only implemented click-through cloaking. The algorithm is presented in major algorithms section.

Detection of the other two cloaking techniques can be implemented easily based on current project. For the second cloaking techniques, we still need to fetch the same web page twice, but with different User Agent field or different IP address. If there is big content difference between two fetches, then server side cloaking is detected. CrawlURI provides interface to set User Agent field. Moreover, the contents can be stored in cloaking cache.

For the third cloaking techniques, since in this project, Heritrix is able to execute JS code, what we could do is just comparing the original web page fetched by fetchHTTP processor to the resulting web page created with JS enabled. Only one fetch on this web page is enough. If simulate HTML events in the web page is optional.

Modifying method updatePrerequisitesCache and cloakingDetection in class DetectCloaking should be able to implement thest two cloaking techniques.

However, individual detection of each cloaking technique may not work, if the spam website use two or more techniques together. For example, a web server may provide different content not only based on User Agent but also based on value of referrer field in HTTP header, so we may not be able to get different web page if only change one of them. And event only one cloaking technique is used, our detection may still fail, because the web server may expect a specific value and provide different content, such as only when referrer = “www.specificvalue.com”, return the real content, since we can only guess that value, the chance of a hit is very small. What we could do is just focuse on the most prevalent cloaking techniques. However, currently we have not yet found any information on prevalence of cloaking techniques. Hopefull we will have it in the future.

. Content Comparison

As we can see, content comparison a must-have step in cloaking detection, no matter how we obtain the contents. In this project, I simply use string comparison method, which means, if there is only one character or digit difference, cloaking is detected. Obviously this simple comparison is not reasonable for cloaking detection. So a more sophisiticated comparison method should be designed. We need to define how much difference or difference in which part within the web page should be considered as cloaking. Currently people have proposed fuzzy hashing (mentioned by Mr. Mohr). In fuzzy hashing, small differences may be ignored. For more information please refer to http://ssdeep.sourceforge.net/ and the full academic paper at
http://dfrws.org/2006/proceedings/12-Kornblum.pdf.

. Dual Fetch

Dural fetch is requried for detection server side cloaking. Doing dual fetch for every web page will not only decrease the performance of Heritrix, but also greatly affect websites which are being crawled. Moreover, spammer can easily notice this “abnormal” behaviors. So far I am not aware of any heuristic method to predict if a website is a spam or not before fetching content and doing further processing. What we could do is sampling the website, and only perform dual fetches on limited number of web pages on one website. A good sampling strategy could be future work. After Spam Detection

Currently, web spam detection module - processor DetectJSRedirection and DetectCloaking, returns true when JS redirection or cloaking is detected, a warining message is logged under debug mode, but no more action will be taken. Logging these information in a log file like crawl.log might be useful.

8 Unsolved Issues

8.1 JUnit Tests

I did not get a chance to write JUnit tests for this project.

8.2 Synchronization in FetchCache and Cache4CloakingDetection

In Heritrxi, there are multiple toeThread to process different uris at the same time, it is possible that fetch cache or cloaking cache is accessed by multiple threads, bad synchronization may result in reading incorrect data or deadlock. In this project, I use Hashtable for all shared data within these two caches, and rough granularity synchronization is implemented. In the future, maybe HashMap could be used and fine granularity synchronization could be done.

9 Usage and Testing

9.1 Code

Source code of this project is avaiable in Heritrxi SVN repository under branches/jenniful_gsoc08/ on sourceforge, and can be checked out at
http://archive-crawler.svn.sourceforge.net/svnroot/archive-crawler/
branches/jenniful_gsoc08/heritrix2/ .

All the packages and classes I developed are in the right places. This project is developed based on Heritrix 2.0, so the code can be run in the same way as Heritrix 2.0 does.

9.2 Crawl Job Profile

In order to test the modules I developed, new job profile need to be created.

Job profile - profile-js-cache-rule-spamdetection (see the attachment) is an example, and ready to use. It is created based on profile-basic_seed_sites, with two caches: FetchCache and Cache4CloakingDetection, four processors: FetchCacheUpdater, ExecuteJS, DetectJSRedirection and DetectCloaking and one MatchesListRegExpDecideRule added in.

FetchCache, FetchCacheUpdater and MatchesListRegExpDecideRule are required for ExecuteJS; DetectJSRedirection and DetectCloaking, and Cache4CloakingDetection is required for DetectCloaking; but DetectJSRedirection and DetectCloaking, and Cache4CloakingDetection do not depend on each other, so they can be added to the profile individually; and MatchesListRegExpDecideRule is used to guarantee that crawl uri with scheme of x-jseval is within link scope.

9.3 Test Cases

Some test cases are attached. There are three sub-directories: testFetchCache, testExecuteJS and testDetectCloaking. All the test cases are simple HTML document, complex testings are not performed.

In **testFetchCache**, tests are used to test functionalities of FetchCache, FetchCache and ExecuteJS. After crawling those test cases, check out the crawl.log file, you will see all .html files are fetched first, and then the external .js files, after that uri with scheme of x-jseval is processed, and finally new links created by JS code are crawled.

test1.html, test2.html, test3.html, test4.html and test5.html are used to test simple JS execution ability including HTML event simulation.

test6.html and test7.html are used to test setTimeout method in JS.

test8.html is used to test the error handling when a resource is required but out of scope. In current project, a logged warning message can be seen under debug mode.

In **testExecuteJS**, tests are also used to test functionalities of FetchCache, FetchCache and ExecuteJS. These test were created during mid-term evaluation, and modified by Mr. Mohr.

case1.html - case6.html are used to test simple JS execution ability including HTML event simulation. In each case, an out link (uri ends with "caseX-success", where X is the number appearing the name of the test case), which cannot be extracted using current released Heritrix, can be discovered by this Heritrix with JS execution ability.

mytest1.html - mytest3.html are to test more on JS execution, and the JS code in these document is more complex.

Set the seed to be the alltests.html, crawler will fetch all the test cases. After crawl job is done, if you check the crawl.log, and see uri ending with "caseX-success", it means all modules worked.

Since almost all test in testFetchCache and testExecuteJS involve JS redirection, they can also be used to test DetectJSRedirection.

In **testDetectCloaking**, tests are used to test DetectCloaking. test9.html has client side click-through cloaking, and testcloaking2.php implements server side cloaking.

Note: since there is no output log file for DetectJSRedirection and testDetectCloaking, the detection results are presented as warning messages, and can only be seen under debug mode.

Acknowledgements

I would like to thank Google and Internet Archive for giving me this opportunity, and thank my mentor Gordon Mohr for his great ideas (the design of fetch cache was proposed by him), valuable comments, advice and his valuable time.

And I would also like to thank people in Lobo project for developing such a nice Javascript-aware and CSS-aware HTML DOM parser - Cobra, and thank Mozilla for JavaScript Engine - Rhino.

References

[1] K. Chellapilla, and A. Maykov. A taxonomy of JavaScript redirection spam. In Proceedings of the 3rd international workshop on Adversarial information retrieval on the web, Banff, Alberta, Canada, 2007.

[2] Z. Gyongyi and H. Garcia-Molina. Web Spam Taxonomy. In 1st International Workshop on Adversarial Information Retrieval on the Web, May 2005.

[3] Y. Niu, Y. M. Wang, H. Chen, M. Ma, and F. Hsu. A Quantitative Study of Forum Spamming Using Context-based Analysis. In Proceedings of Network and Distributed System Security (NDSS) Symposium, February 2007.

[4] Rhino, http://www.mozilla.org/rhino/

[5] Cobra, http://lobobrowser.org/cobra.jsp

A little About Me

My name is Ping Wang, and I am a Ph.D. student majoring Computer Science at University of Central Florida.

My email address is pwang at cs dot ucf dot edu. Please feel free to contact me, if you have any comments or questions regarding this project.