A Dynamic Internet: Categorizing Webpage Content Changes over Time

ECE8813 Advanced Network Security

Section A, Team Tyler
Project Advisor, Dr. Manos Antonakakis

Tyler Lisowski
Kim Yie
Yakshdeep Kaul
Judah Okeleye
Zongwan Cao

Submitted

November 18th , 2015

Table of Contents

Abstract

The HTML Document Object Model (DOM) is the standard for web page design. Good programming practice follows that any change, addition, or deletion of HTML elements will adhere to this standard. Hence it is necessary that any change to a webpage, be it benign or malicious, will follow the DOM standard to ensure proper validation by web browsers.

In this paper, we propose a system that takes advantage of the HTML DOM tree to classify changes over time to a webpage as malicious or benign. We implemented an HTML DOM tree Diff Engine in Python that can detect changes between two webpages. Our system uses a threaded web crawler to crawl the Alexa Top 1 Million Webpages and uses a machine learning classifier to distinguish between malicious and benign changes that occur in two historical versions of the same webpage.

A Dynamic Internet: Categorizing Webpage Content Changes over Time

1. Introduction

HTML is used by many websites to create webpages that are visited every day. Some webpages change every minute, while others remain the same for months. Due to various reasons (XSS, SQL injection, etc.), some changes introduced to the webpage may cause the webpage to have malicious behavior. This could happen when an attacker injects malicious content or modifies existing HTML content in a malicious way. If this change causes something malicious to be downloaded or ran in the background then we define that change as a malicious change.

In this paper we introduce a classifier that performs content analysis on a webpage. Based on the change from the previous historical copies of the webpage, the classifier determines if a change introduced to the webpage was benign or malicious. Our system could be used in a monitoring environment by web admins. For example, take a site that has been the victim to a XSS injection attack. Our classifier would detect that the page was changed in a malicious way from the previous version that was hosted by the website. Another environment the classifier could run in is alongside a browser, such as Google Chrome. This would allow all users of the browser to be notified that a previously benign site has turned into a malicious one because of a change in the web content of the page.

This paper presents the following contributions:

- **HTML Content Analysis**: We develop a classifier that performs content analysis on a webpage to determine malicious changes on the webpage. The classifier is validated using 10-fold cross validation and performs at a high TP rate of 97.3% with no false positives.
- **HTML Differ**: The classifier uses results from a novel differ algorithm. This algorithm detects additions to the DOM tree in addition to changes in attribute values in HTML tags. The differ algorithm was successfully implemented in Python 2.7.

The remainder of the paper is structured as follows: Section 2 provides related work. Section 3 details the datasets used in the overall implementation of the system. Section 4 gives an in-depth explanation of how the system is implemented, which includes the web crawler, Diff Engine, and the machine learning classifier. Section 5 describes our results, and we finally conclude in Section 6.

2. Background/Related Work

There is some existing work related to HTML webpage change detection. Hassan Artial e.t. proposed a fast HTML web page detection approach that saves computation time by limiting the similarity computations and by leveraging certain hashing techniques [1]. Imad Khoury e.t. described an efficient webpage change detection system based on an optimized Hungarian Algorithm [2]. However, neither of the authors mentioned above attempted to analyze or classify detected changes as benign or malicious. Their work is about how to improve the detection algorithm.

Some other researchers focused on detection and analysis of malicious webpages. Christian Seifert e.t. presented a classification method for detecting malicious webpages that involves the underlying static attributes of the initial HTTP response and the HTML code [3]. Marco Cova proposed an approach to the detection and analysis of malicious JavaScript code [4]. However, they focused on the entire webpage and HTTP response as a whole, compared to specifically examining the changes over time in a webpage.

Our contribution is to detect the changing part of the same website over time, then classify these changes as benign or malicious using a machine learning classifier. In this way, we can have a better understanding of the webpage change in a security view.

3. Datasets Used

3.1 Page Content

Our dataset consists of historically fetched pages from the Alexa Top 1 Million from October 28th, 2015 to November 16th, 2015. Pages are fetched every 6 hours at the following time intervals: 4 AM EST, 10 AM EST, 4 PM EST, and 10 PM EST. This is done to ensure we can see changes occur on a daily basis (possibly even multiple times per day). It also can be used to provide insight on when webpages tend to change. For example, real time news websites like CNN will be updating at a more frequent interval than a banking website like Bank of America. The set from each fetch period consists of 10,000 total pages. The first 5,000 correspond to the first 5,000 Alexa Top 1 Million Webpages according to popularity. The next 5,000 start at the $150{,}000^{th}$ most popular page and go to the $155{,}000^{th}$ most popular page. This choice is made based on the fact that pages lower in the popularity ranking are more likely to be targeted and compromised due to the likelihood of them having more vulnerabilities than pages at the top of

the Alexa Top 1 Million List (Google is way less likely to have a vulnerability than autzon.com). However, these web pages still generate enough traffic to be a desirable target for an attacker. This gives us a dataset in which we hope to have websites that consistently change in a non-malicious way (the top 5000 web pages) and a set that will contain some web pages that go through malicious changes and then cleanup cycles (150,000-155,000th most popular webpages). The raw HTML content of each page is stored. We later aim to use this content to train and test a machine learning classifier to classify the change cycles these webpages go through.

3.2 IP Whois Data

We use an IP Whois database to lookup IP information for the IPs that the domains resolve to [5]. With this data, we can analyze historical data about what IPs and Autonomous System Numbers the server resolved to in the past and use this data to classify changes. For example, malicious changes might only occur when the resolved IP is a specific IP because the attacker is targeting a specific geographic region. Therefore, this data can help us tie together changes that happen with web content along with changes that happen over time with the resolved IPs. We start with a given domain that we have fetched the page contents for. We then see what IP this domain resolves to on our machine and query the IP Whois database for data around the IP address. An example output of information for an IP address is shown in Figure 1. This consists of the following information:

- asn- the Autonomous System Number (ASN)
- asn_registry- the registry that the Autonomous System Number is registered to
- asn_country_code- country code for the ASN
- resolved_ip- the actual IP that the webpage resolved to
- asn_cidr- the Classless Interdomain Routing address of the ASN

```
{"asn_registry": "arin",
"asn_date": "2007-03-13",
"asn_country_code": "US",
"resolved_IP": "74.125.21.113",
"raw": null,
 "asn_cidr": "74.125.21.0/24",
"raw_referral": null,
"query": "74.125.21.113",
"referral": null,
"nets": [{"updated": "2012-02-24T00:00:00", "handle":
"NET-74-125-0-0-1", "description": "Google Inc.", "tech_emails":
"arin-contact@google.com", "abuse_emails": "arin-contact@google.com",
"postal_code": "94043", "address": "1600 Amphitheatre Parkway",
"cidr": "74.125.0.0/16", "city": "Mountain View", "name": "GOOGLE",
"created": "2007-03-13T00:00:00", "country": "US", "state": "CA",
"range": "74.125.0.0 - 74.125.255.255", "misc_emails": null}],
"asn": "15169"}
```


Figure 1. Real example of results from the ipwhois Python library.

To provide some background, an Autonomous System is a range of IP addresses grouped together in the BGP protocol [6]. This range of IP addresses is assigned a unique global number called an Autonomous System Number [6]. In order to obtain this, the corporation (ISPs, Google, etc.) needs to go register the Autonomous System Number with a registry [6]. An example registry would be the American Registry for Internet Numbers. These registries keep track of the information specified above and users can query their database for the information.

3.3 Google Safe Browsing Lookup

Google uncovers about 9,500 new malicious websites every day [9]. The Google Safe Browsing Lookup API permits applications to verify URLs against this constantly updated list of suspected phishing, malware, and unwanted software pages. We understand that such lists are reactive but we use the Safe Browsing List only as an added layer of verification to reduce the false positives that our system may generate. We use the Safe Browsing Lookup Library for Python to fetch the response (limited to 10,000 lookups per day) pertaining to a particular webpage.

The lookup requires the need of an API key, which a user can obtain only after registering with Google, and varies based on the type of lookup required. Given below is an example of a client's HTTP request for a suspected malicious page and the corresponding answer by the server:

Clients request URL:

```
https://sb-ssl.google.com/safebrowsing/api/lookup?client=demo-
app&key=12345&appver=1.5.2&pver=3.1&url=http%3A%2F%2Fianfette.org%2F
```

Server's response body:

```
malware
```

4. Implementation

In this section, we describe in detail how our system works and how the datasets mentioned in Section 3 are used. We start by introducing the threaded web crawler that fetches a range of webpages from the Alexa Top 1 Million List and collects the IP Whois data of the resolved IPs. Once the crawler assembles the dataset of webpages, it is then fed to the Diff Engine. The Diff Engine parses and compares two copies of the same webpage collected at different intervals of time, according to the Document Object Model (DOM) for valid HTML. The output of the Diff Engine (addition/deletion of nodes, text changes, etc.) is used to train and test the machine learning classifier. We now describe each of these at greater length.

4.1 Web Crawler

We built a threaded web crawler in Python to handle fetching the page content and IP Whois data of the resolved IPs from the Alexa Top 1 Million List. Figure 2 shows the workflow the web crawler executes. The web crawler starts by reading from the Alexa Top 1 Million CSV file and fetching webpages a page at a time. The requests python library is used to handle the fetching process [7]. After a page is fetched, the raw page is stored in a python dictionary, with the domain name as the key and the page content as the value. Next, the IP Whois data is fetched for the resolved IP address and stored in a similar python dictionary (key is domain and value is the JSON output from the IP Whois database). A checkpoint is set based on the total number of fetched pages. At every checkpoint interval, the web data is then serialized to files in the following format: '<domainName>.html'. The files are all stored under the 'web/' directory and are then stored in a subdirectory of the 'web/' directory that corresponds to the 6 hour time period in which the web page was fetched. This subdirectory is named in the following format: '<year>_<month>_<day>_<hour>/'. The IP Whois data is stored under the ip/ directory and follows the same format for the file and subdirectory, but is stored in a JSON format.

The execution process is handled by spawning a thread per page fetch. The maximum number of active threads (and therefore active outstanding network connections) is limited to a threshold value. This threshold value is chosen based on the network resources we have available

to us. It can be adjusted for different network conditions and available hardware.

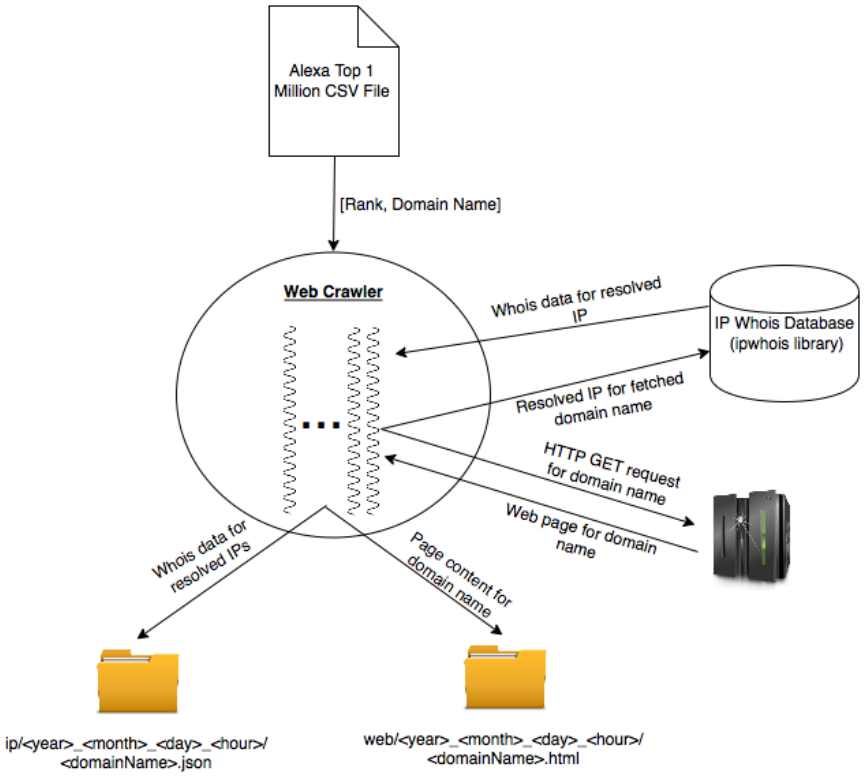

Figure 2. Web crawler implementation.

4.2 Diff Engine

Our Diff Engine uses a hashing function to map each node in the HTML DOM tree with a specific key. Based on each node's unique key, the difference between nodes in the old file and the new file can be detected.

4.2.1 Key Generation

For each node, we set its key as: documentNode.tag'/!!/'parentNode[n].tag'/!!/'….'/!!/' parentNode[k].tag'/!!/….parentNode[1].tag'/!/'node.tag_i. '/!!/' and '/!/' are used as separators and the whole string can be viewed as an index into the HTML document that leads to where the element resides. Here n is at least 1, because every element will be inside the <html> tag. ParentNode[1] is the immediate ancestor of the current node (parentNode[k+1] is ancestor node of parentNode[k] and so on). '_i' is an optional suffix to distinguish sibling nodes, namely to differentiate nodes which have the same DOM tree path. Among sibling nodes, the first one we visited does not have "_i" in

its key, the second one has "_1", the third one has "_2", and so on. Key generation is done for both historic copies of the web page.

For example, given the following piece of HTML:

```
<html>
    <body>
        <script> ……. </script>
        <script> ……. </script>
    </body>
</html>
```

The key for the first <script> element is '[document]/!!/html/!!/body/!/script'. Second <script> element's key is '[document]/!!/html/!!/body/!/script_1'.

4.2.2 Locating Corresponding Nodes

The goal of our Diff Engine is to find if: 1) a node is modified, 2) a node is deleted, and 3) a node is added. To accomplish this, we compare the original node with its corresponding node in the new file. If node A in the original file has the same path as node B in the new file, then A corresponds with B or B's siblings. For example, [document]/!!/html/!!/body/!/script in the original file could correspond to either [document]/!!/html/!!/body/!/script or [document]/!!/html/!!/body/!/script_1 in the new file if the new file contains both of these elements. We calculate the similarity between A and all nodes with the same key in the new file using the lexicographic distance of the string representation of both nodes. We can get similarity score between node A and all corresponding nodes in the new file. Then among the corresponding nodes in the new file, we pick the one with the highest similarity score to node A as the corresponding node to compare node A to.

4.2.3 Find Added Nodes and Deleted Nodes

After locating the most similar corresponding nodes in the new webpage to the nodes in the initial webpage, we may have remaining nodes that did not get matched. All corresponding nodes are formed from modifications of the original nodes in the original webpage. However, the remaining nodes in the original file are those who cannot be paired with nodes in new file, which means those nodes are deleted in the new file. We classify these nodes as deleted nodes. The

opposite holds true for the new file: nodes in the new file that do not have corresponding nodes in the original file are new nodes and we classify them as such.

4.2.4 Output Results

For node pair A and B, we detect no changes if its similarity score is 1 since the nodes are the same. If the pair's similarity score is less than 1, we then compare A with B. We examine both text changes and attribute changes in leaf nodes but only focus on attribute changes in parent nodes. Text changes are disregarded in parent nodes to reduce redundancy in the output because text changes inside the parent node will already be captured in changes to the leaf nodes. For attribute changes we have three types: attribute value change, added attribute, and deleted attribute. For text changes there are also three types: insert, delete and replace. We store all of the results in a JSON format. We format the information about modified nodes in the following key-value pairs: (1) 'afterText': the text that appears in new node, (2) 'elementType': the type of the node, (3) 'fullText': the combination of text in the old node and the new node, (4) 'op': the type of change (insert, delete or replace), (5) 'other info': the type of change (attribute value change, added attribute, deleted attribute, or text change), and (6) 'raw change': the change that resulted in the new node.

For added and deleted nodes, we create the following key-value pairs: (1) 'afterAttribute': the attributes of added or deleted node, (2)'afterText': the text of the added or deleted node, (3) 'elementType': the type of the node (script, iframe, etc.), (4) 'other info': whether the node was added or deleted. The output in Figure 3 is from changing:

Node A: <script src="/js/cookies/src/jquery.cookie.js"></script> to

Node B: <script src= "cookies/src/jquery.cookie.js"></script>

```
{'afterText': 'cookies/src/jquery.cookie.js',
 'elementType': 'script',
 'fullText': '<del>/js/</del>cookies/src/jquery.cookie.js',
 'op': 'del',
 'otherInfo': 'ATTRIBUTE VALUE CHANGE',
 'rawChange': u'/js/'},
```

Figure 3. Example output for an attribute value change in the new webpage.

The output for adding a new node <script src= "www.google.com"></script> is shown in Figure 4.

```
['afterAttribute': {'src': 'www.google.com'},
 'afterText': None,
 'elementType': 'script',
 'fullText': '',
 'op': '',
 'otherInfo': 'ADDED NODE',
 'rawAttributeChange': {'src': 'www.google.com'},
 'rawTextChange': None}}
```

Figure 4. Example output for an added node in the new webpage.

4.3 Classifier

After calculating the set of changes between a pair of webpages, each change can be passed to a classifier that will output a classification based on whether the change exhibits properties that are characteristic of malicious or benign webpage changes. The classifier is trained on the output of the diffing engine and also on features of web content.

4.3.1 Training Dataset

For our malicious webpage change training dataset, we used drive-by malicious JavaScript examples as flagged by the 'malware-traffic-analysis.net'[8] blog between the months of July 2015 to November 2015. For each entry in the blog, we found and recreated the change necessary to model an attacker that has injected the drive-by script into the webpage. The drive-by injections mostly consist of malicious JavaScript and iframe injections. For our benign training dataset, we examine changes from cnn.com because it is a reputable news site that is likely to have daily updates and not to have been compromised. This gives us a training set consisting of 650 benign changes and 74 malicious changes.

4.3.2 Feature Selection

Our feature selection shares many similarities to previous approaches to classifying malicious webpages. However, since we have used an HTML diffing engine, we receive the benefits having filtered the set of inputs to the classifier to only whether new changes are malicious or not. Our set of features is as follows:

- **Feature 1:** *elementType.* We extract the tag of the html change, whether it is a script, iframe, div, or p tag. We create an enumerated lookup table to translate between the nominal tag name to a numerical tag value.
- **Feature 2:** *editType.* The edit type can be an inserted node or a deleted node. These options are translated to numerical values of 0 and 1 for the classifier, respectively.
- **Feature 3:** *scriptLen.* The length of the script is calculated. If the change is not a script, the value is assigned to -1.
- **Feature 4:** *specialCharRatio.* The ratio of special characters to regular ascii characters are calculated. We find that many malicious payloads are obfuscated with the use of many special characters and punctuation that benign changes do not exhibit.
- **Feature 5:** *GSB.* Many malicious changes use the 'src' attribute to specify a link to a payload. For each change that specifies a source, we pass it to the GoogleSafeBrowsing API to see if Google has already blacklisted that domain or not.
- **Feature 6:** *jsEval.* Many malicious changes use the eval, document.write, or document.createElement to deobfuscate or trigger vulnerabilities to a website during runtime. The number of occurrences of these JavaScript functions is recorded as a feature for the classifier.

4.3.3 Classifier Detection Results and Model Selection

For the classifier model, we trained a classifier using the following algorithms and tested it with 10-fold cross validation. We ultimately choose the classifier with the best results.

1) Naive Bayes

The Naive Bayes algorithm performed with a TP rate of 81.1% and a FP rate of 1.8%. The ROC is shown in Figure 5.

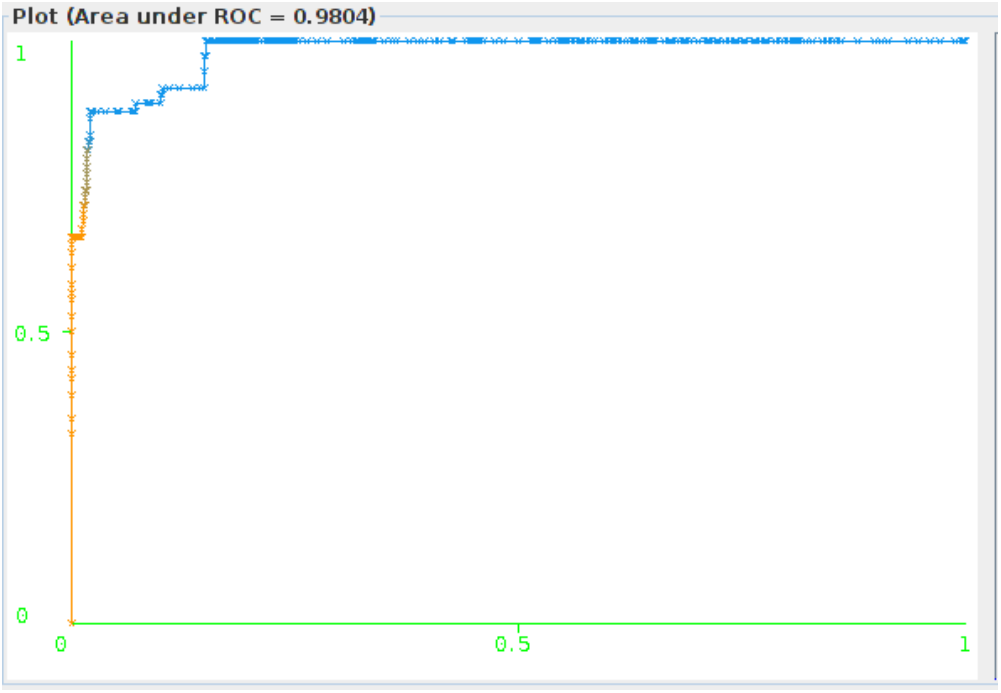

Figure 5. The ROC curve reported by Weka for the Naive Bayes classifier.

2) J48

A J48 Classifier performed with TP rate of 97.3% and FP rate of 0%. The ROC curve is shown Figure 6.

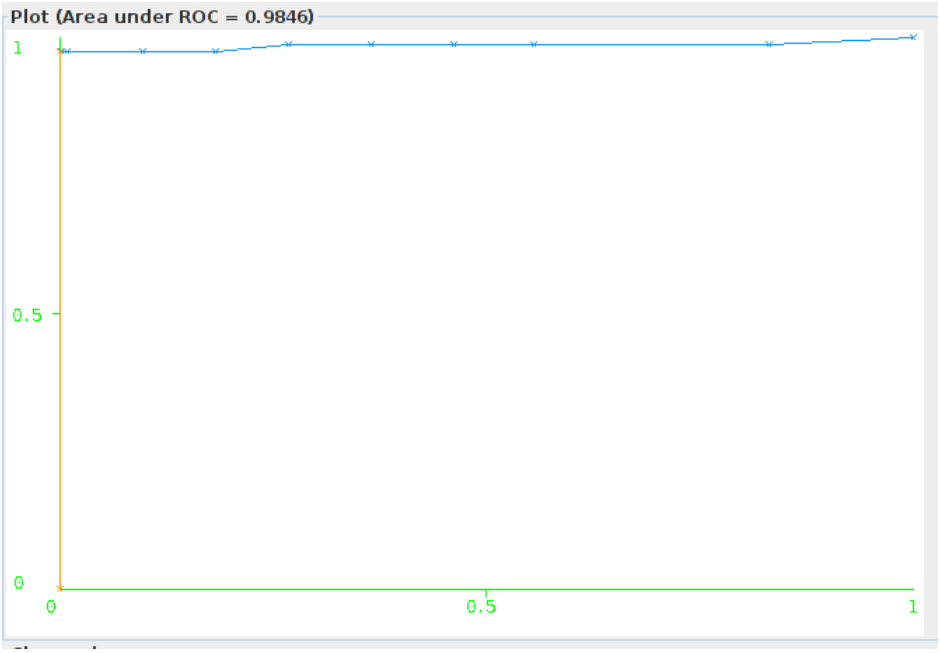

Figure 6. The ROC of the J48 classifier algorithm reported by Weka.

3) Random Forest

The Random Forest Classifier performed with a TP rate of 97.3% and a FP rate of 0%. The ROC curve is shown Figure 7.

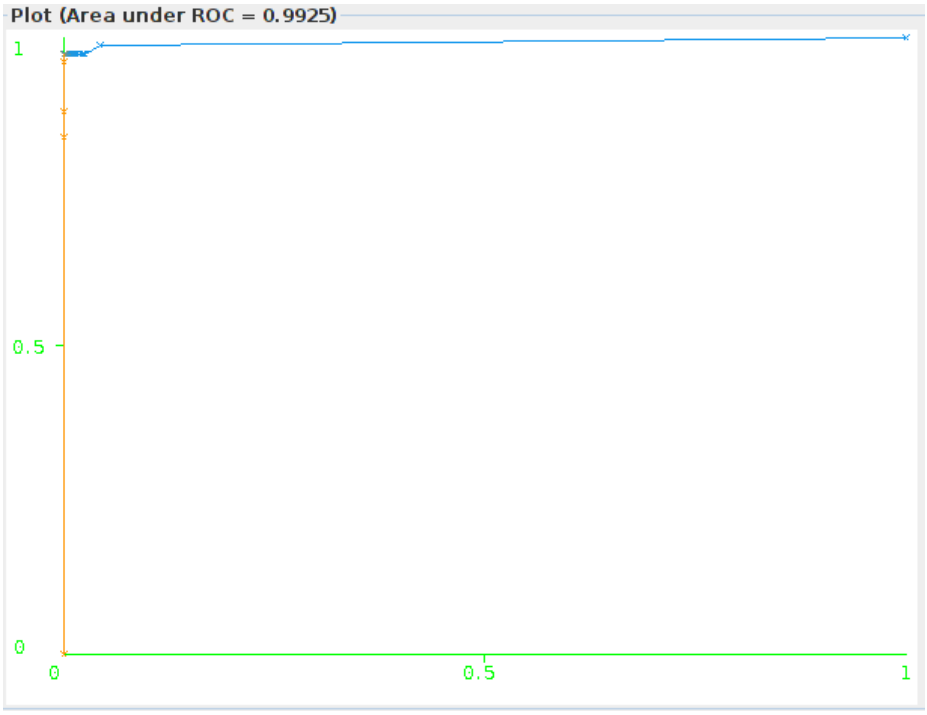

Figure 7. The ROC of the Random Forest classifier reported by Weka.

4.3.4 Feature Ranking and Feature Selection

We run the Information Gain attribute evaluator on these attributes. The attributes that provide information ranked most to least are: editType, scriptLen, specialCharRatio, GSB, elementType, and jsEval. The ranked attribute scores reported by Weka are shown in the table below.

Attribute Name	Ranked Value
editType	0.3319
scriptLen	0.3033
specialCharRatio	0.2987
GSB	0.2629
elementType	0.2542
jsEval	0.0399

Table 1. A table with reported InfoGain attribute evaluator scores for each feature.

When jsEval is removed from the feature set, the J48 tree performs at 97.3% TP and 0% FP rate still. When both jsEval and elementType are removed from the feature set, the J48 tree performs with 87.8% TP and 0% FP rate. So, our final feature set for the classifier can have the jsEval feature removed from the set.

5. Results and Discussion

Our results of each classifier using 10-fold cross validation are very promising, each leading to high true positive up to 97.3% and no false positives. Despite the good performance of the classifier on the test dataset, we suspect that there should be additional features that can be used, such as using domain name reputation on src attributes of any new HTML tags. Many of the malicious JavaScript drive-by examples identified by malicious-traffic-analysis.net blog share many similar characteristics, perhaps because the author of the blog tracks several of the same actors in their use of drive-bys to download their exploit kit. In the future, malicious web page changes should be gathered from multiple different sources. Finally, a fuller feature set may include some dynamic execution of extracted JavaScript differences in a JavaScript Engine in order to unwrap obfuscated JavaScript payloads.

One consideration that needs to be taken into account is the scalability of our system. Our web crawler scales well with the Alexa Top 1 Million Websites. In a timed test of fetching the top 10,000 Alexa 1 Million Webpages, the crawler can fetch at a rate of 2,500 pages per minute on one virtual machine with a 10 Gbit/s connection when just fetching web content. It slows down to approximately 100 pages per minute with one virtual machine when fetching the IP Whois data for these domains in the same test, but this data can be fetched independent of the

time sensitive web content as long as we store the resolved IP. Therefore, the main time sensitive bottleneck will be fetching the web content. Adding a load balancer and distributing the fetching workload across multiple nodes can speed up the fetching process. However, the Diff Engine takes up to several minutes per page due to the lexicographic comparison of the string representation of nodes. Additionally, one of the features used in the classifier requires us to use Google's GSB service. By default, the GSB provides an API key that allows 10,000 lookups a day, and getting an extension to that limit requires contacting Google.

5.1 Evasion

There are several methods to evade the classifier we have trained. One method is to avoid creating malicious changes that target the features we use. The features that would be most effective to evade our classifier model would be the GSB, specialCharRatio, and scriptLen features. An attacker can query the GoogleSafeBrowsing API to determine if a domain is flagged before using it in an attack. The scriptLen and specialCharRatio features can be avoidable by using simpler (less obfuscated) payloads, but the tradeoff for the attacker is that simple signature based detection more likely to detect less obfuscated payloads. The elementType and editType features are not evadable, because they are intrinsic characteristics of any new JavaScript or iframe injection into a compromised website.

6. Conclusion

In this paper we created a classifier that can detect malicious HTML changes on a webpage. It uses a unique diffing algorithm to detect key changes, as opposed to the simple text diffing. Our results show that it is possible to train a classifier to detect malicious changes in historical versions of webpages with a high true positive detection rate of 97.3% and no false positives.

7. Acknowledgement

We would like to thank everyone who was involved in guiding and helping us throughout this project, especially Professor Manos Antonakakis and his PhD students. This work was done as a semester long project for the course Advanced Computer Security taught by Professor Manos Antonakakis, and offered at Georgia Institute of Technology in Fall 2015.

8. References

[1] H. Artail and M. Abi-Aad,“An enhanced web page change detection approach based on limiting similarity computations to elements of same type”, Springer Science + Business Media, LLC, pp 1-21, 2007
http://www.sciencedirect.com/science/article/pii/S0169023X08000414

[2] Imad Khoury, Rami M. El-Mawas, Oussama El-Rawas, Elias F. Mounayar, and Hassan Artail, “An Efficient Web Page Change Detection System Based on an Optimized Hungarian Algorithm ”, In IEEE Transactions on Knowledge and Data Engineering, Vol. 19, NO. 5, pp 599-613, May 2007.
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4138199

[3] Christian Seifert,Ian Welch,Peter Komisarczuk,”Identification of Malicious Web Pages with Static Heuristics”,CiteCeer, Proceedings of the 43rd Hawaii International Conference on System Sciences 2010

[4] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In Proceedings of the 19th international conference on World Wide Web (WWW), pages 281–290, 2010

[5] P. Hane, “ipwhois- ipwhois 0.11.0 documentation”, Phillip Hanem pp 1, 2015
http://http://secynic.github.io/ipwhois/

[6] G. Huston, “Exploring Autonomous System Numbers”, Cisco Systems, pp 1-10, 2006
http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_91/autonomous_system_numbers.html

[7] K. Reitz, “Requests: HTTP for Humans”, Kenneth Reitz, 2015 http://docs.python-requests.org/en/latest/

[8] Brad, “A source for pcap files and malware samples…”, Malware-Traffic-Analysis.net, 2015
http://www.malware-traffic-analysis.net/2015/index.html

[9] N. Provos, Safe Browsing - Protecting Web Users for 5 Years and Counting, Google Online Security Blog, 2012 http://googleonlinesecurity.blogspot.com/2012/06/safe-browsing-protecting-web-users-for.html